Évolution dynamique des systèmes d'exploitation, une approche par la programmation par aspects

Nicolas Loriant

UNIVERSITÉ DE NANTES
FACULTÉ DES SCIENCES ET DES TECHNIQUES

**ÉCOLE DOCTORALE SCIENCES ET TECHNOLOGIES
DE L'INFORMATION ET DES MATÉRIAUX**

Année 2007

N° attribué par la bibliothèque

Évolution dynamique des systèmes d'exploitation, une approche par la programmation par aspects

---

THÈSE DE DOCTORAT

Discipline : Informatique

Spécialité : Informatique

présentée et soutenue publiquement par

Nicolas Loriant

le 7 décembre 2007 à l'École Nationale Supérieure des Techniques Industrielles et des Mines de Nantes

devant le jury ci-dessous

Président	:	Jean-Bézivin, Professeur	Université de Nantes
Rapporteurs	:	Daniel Hagimont, Professeur	Institut National Polytechnique de Toulouse
		Lionel Seinturier, Professeur	Université de Lille 1
Examinateurs	:	Jean-Marc Menaud, Docteur	École des Mines de Nantes
		Christine Morin, DR	INRIA Rennes
Directeur de thèse	:	Gilles Muller, Professeur	École des Mines de Nantes

Directeur de Thèse : Gilles Muller
Responsable Scientifique : Jean-Marc Menaud

N° ED 366-332

ÉVOLUTION DYNAMIQUE DES SYSTÈMES D'EXPLOITATION, UNE APPROCHE PAR LA PROGRAMMATION PAR ASPECTS

Runtime Evolution of Operating Systems using Aspect-Oriented-Programming

Nicolas Loriant

Université de Nantes

Remerciements

Je souhaite commencer par remercier Pierre Cointe pour m'avoir accueilli au sein de l'équipe Obasco et pour son implication dans mon travail. Je remercie Gilles Muller d'avoir accepté d'être mon directeur de thèse et Jean-Marc Menaud pour m'avoir accompagné au quotidien dans son déroulement. Je souhaite les remercier tous les deux, pour m'avoir permis par leur encadrement, de gagner l'autonomie nécessaire au parcours de thèse. Je saisi également l'opportunité de remercier les membres de mon jury, dont les commentaires ont permis d'ouvrir de nouvelles perspectives sur mon travail. Je remercie Thomas Ledoux qui m'a fournit les moyens de terminer cette thèse dans de bonnes conditions et pour m'avoir proposé un nouveau projet de travail enrichissant. Je remercie également Julia Lawall pour la rigueur de ses relectures et l'aide qu'elle m'a apportée tout au long de ma thèse.

Il m'est difficile de remercier toutes les personnes grâce auxquelles j'ai pu mener à terme ces trois années de recherche. Bien plus qu'une liste de noms, ces personnes ont supporté mon caractère difficile et rendu mon parcours de thèse agréable et enrichissant. Je leurs en suis redevable.

À Marc Ségura-Devillechaise et Thomas Fritz, je n'aurais rien fait sans eux. À un cordonnier de la recherche, Mister D.

À ceux avec qui j'ai partagé le banc de nage pendant plus de trois années : Xavier Lorca, Christophe Augier, Richard Urunuela et Florian Minjat. Aux grouillots, lapins et autres strafe jumper wanabees : Pierre Lavoix, Guillaume Richaud, Fabien Hermenier et Hadrien Cambazard.

À Sébastien, Caroline et Adélaïde.

À toute ma famille, mes parents, mamie et Titine, dont l'affection et le soutien m'ont permis de ne pas faillir.

Sommaire

Sommaire i

Table des figures iii

Liste des tableaux iv

Liste des listings v

1 Introduction 1
- 1.1 Objectifs . . . 2
- 1.2 Contributions . . . 3
  - 1.2.1 Langage d'aspect . . . 3
  - 1.2.2 Arachne . . . 3
  - 1.2.3 Applications . . . 4
- 1.3 Organisation du document . . . 4
- 1.4 Diffusion scientifique . . . 5

I Contexte de l'Étude 7

2 Contexte de l'étude 9
- 2.1 Designs extensibles . . . 10
  - 2.1.1 La paramétrisation . . . 11
  - 2.1.2 Le chargement dynamique – plugins . . . 11
  - 2.1.3 La spécialisation de programmes . . . 12
  - 2.1.4 Architectures extensibles . . . 12
- 2.2 Environnements d'exécution pour logiciels extensibles . . . 13
  - 2.2.1 Programmation par composants . . . 13
  - 2.2.2 Composition de méta-niveaux . . . 14
  - 2.2.3 Environnements d'exécution dédiés . . . 15
  - 2.2.4 L'interposition . . . 16
- 2.3 Transformation de programmes . . . 17
  - 2.3.1 Les systèmes de réécriture de code . . . 18
  - 2.3.2 La programmation par aspects . . . 22
- 2.4 Récapitulatif . . . 24
- 2.5 Proposition – Programmation par aspects par réécriture de code . . . 25

3 Problèmes illustrateurs **27**
3.1 Correction d'erreurs – Trous de sécurité 28
3.1.1 Exploitations de l'allocateur dynamique de mémoire 31
3.1.2 Débordements de tableaux 34
3.1.3 Interblocages de verrous 35
3.2 Ajout d'une politique de préchargement dans Squid 36
3.3 Adaptation 37
3.3.1 Supervision de l'exécution de Squid 37
3.3.2 Remplacement de TCP par UDP 38
3.4 Bilan 39

II Contribution **41**

4 Un langage pour l'adaptation **43**
4.1 Choix des points de jonction 44
4.1.1 La relation code C – code machine 45
4.1.2 Les points de jonction 47
4.1.2.1 Événements du code 47
4.1.2.2 Manipulations de données 47
4.1.2.3 Inspection de l'historique 48
4.2 Le langage de coupe 48
4.2.1 Sélecteurs de points de jonction 49
4.2.2 Variables de coupe 49
4.3 Le langage d'aspect 50
4.3.1 Aspects 50
4.3.2 Placement des aspects 51
4.4 Exemples illustrateurs 52
4.4.1 Concepts de base – Exploitation de l'allocateur de mémoire C . . 53
4.4.2 Les flots de contrôle – Préchargement dans le cache Web Squid . 53
4.4.3 Les séquences – Remplacement de TCP par UDP 54
4.4.4 Les accès aux variables – Les débordements de tableaux 55
4.4.5 *bind* – Les interblocages de verrous 56
4.4.6 Placement des aspects – La supervision du cache Web Squid . . 57
4.5 Conclusion 57

5 Arachne – une mise en œuvre modulaire **59**
5.1 Vue d'ensemble 60
5.1.1 Le compilateur d'aspect 61
5.1.2 Le tisseur d'aspect 61
5.1.3 Organisation fonctionnelle 62
5.1.4 Extensibilité 62
5.2 Injection du noyau de réécriture 63
5.3 Interception du flot d'exécution 64
5.3.1 Technique de réécriture des points de jonction 64
5.3.1.1 Réécriture sûre d'un point de jonction 65
5.3.2 Les crochets – cohérence du déclenchement des aspects 66

5.4 Compilation des aspects . . . 67
5.4.1 Le code exécutable des aspects . . . 68
5.4.1.1 Aspects sur les appels de fonction . . . 69
5.4.1.2 Aspects sur les flots de contrôle . . . 69
5.4.2 Les directives de tissage . . . 70
5.4.2.1 Aspects sur les appels de fonction . . . 70
5.4.2.2 Aspects sur les flots de contrôle . . . 71
5.5 Résolution des informations de tissage . . . 71
5.5.1 Résolution des symboles . . . 72
5.5.2 Localisation des points de jonction . . . 73
5.6 Conclusion . . . 74

6 Expérimentations et évaluations 75
6.1 Protocole expérimental . . . 76
6.1.1 Difficultés . . . 76
6.1.2 Protocole . . . 77
6.1.3 Validation . . . 78
6.2 Micro-évaluations . . . 78
6.3 Macro-évaluations . . . 81
6.3.1 Ajout d'une politique de préchargement dans Squid . . . 81
6.3.2 Corrections de trous de sécurité . . . 83
6.3.2.1 Configuration des *patchs* . . . 84
6.3.2.2 Faisabilité . . . 84
6.3.2.3 Mesures de performance . . . 85
6.3.3 Supervision des accès aux systèmes de fichiers . . . 86
6.4 Conclusion . . . 87

7 Conclusion 89

Bibliographie 93

Glossaire 103

Table des figures

2.1 Organisation de l'état de l'art . . . 10
2.2 Organisation du protocole de méta-objet dans Apertos . . . 15
2.3 Adaptation par interposition avec SLIC . . . 17
2.4 SLIC : performances . . . 17
2.5 La préoccupation logging dans Tomcat . . . 23

3.1 Rapports d'incidents de sécurité entre 1988 et 2002 29
3.2 Vulnérabilités constatées entre 1995 et 2002 29
3.3 Vulnérabilités dans les logiciels libres 31
(a) Répartition par langages de programmation 31
(b) Répartition par types de vulnérabilités dans les logiciels écrits en C . 31
3.4 Réalisation d'une attaque de type « double free bug » 33
(a) Cinq fragments consécutifs, le premier et le dernier sont libres 33
(b) Après libération du fragment F 33
(c) Après une seconde libération de F 33
(d) Après allocation du fragment F 33
(e) Préparation de la charge 33
(f) Une nouvelle allocation de F finit la mise en place de l'attaque 33
3.5 Exploitation d'un débordement de tableau 35
(a) Avant attaque 35
(b) Après injection de la charge 35
3.6 Utilisation des interfaces TCP et UDP 39

4.1 Le modèle de points de jonction d'Arachne 47
4.2 Le langage de coupe d'Arachne 50
4.3 Le langage d'aspect d'Arachne 52

5.1 Architecture générale du système Arachne 60
5.2 Fonctionnement des crochets 67

6.1 Performance des contructions *seq*, *control flow* et *readGlobal* 81
6.2 Résultats de la localisation des vulnérabilités 85

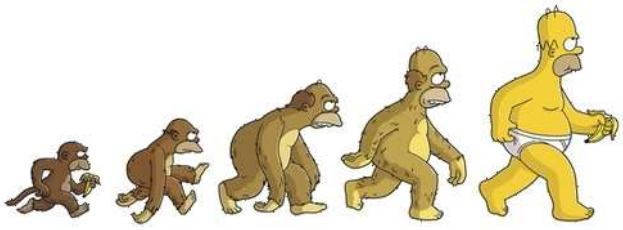
Liste des tableaux

2.1 Récapitulatif par approche 25

6.1 Coûts du déclenchement des aspects 80
6.2 Évaluation de l'implémentation par aspect d'une politique de préchargement 82
6.3 Mesure de PolyMix-4 pendant les deux pics de requêtes 83
6.4 Mesures sur l'utilisation des disques dans Squid 86

Liste des listings

2.1 Optimisation des branchements inconditionnels consécutifs avec Vulcan . . 19
2.2 Génération de l'expression $A = (B * C + D)/(E - F)$ par l'API Kerninst . 19
2.3 Insertion d'un compteur d'appels de fonction avec Kerninst 20
2.4 SystemTap, affichage des 20 appels systèmes les plus utilisés 22
3.1 Structures de l'allocateur mémoire 32
4.1 Un aspect pour détecter les « double free bugs » 53
4.2 Un aspect pour ajouter le pré-chargement dans Squid 54
4.3 Un aspect pour passer de TCP à UDP 55
4.4 Un aspect pour détecter les débordements de tableaux 55
4.5 Un aspect pour détecter les « deadlocks » 56
4.6 Un aspect pour la supervision de Squid 57
5.1 Un patron de crochet pour des aspects sur appels de fonction 67
5.2 L'aspect « exemple » intercepte les appels à la fonction *foo* 69
5.3 Code généré pour l'aspect « exemple » du listing 5.2 69
5.4 Un aspect sur flot de contrôle 70
5.5 Code généré pour l'aspect « exemple » du listing 5.4 70
5.6 Directives de tissage pour l'aspect « exemple » du listing 5.2 71
5.7 Directives de tissage pour le flot de contrôle « exemple » du listing 5.4 . . . 71
6.1 Intructions de mesure des cycles écoulés et de sérialisation 78
6.2 Protocole expérimental pour la mesure de durées en cycles processeur . . . 79



Chapitre 1

Introduction

Pendant des siècles, la culture judéo-chrétienne a répandu la vision fixiste[1] de la vie au travers de l'« Ancien Testament ». La pensée dominante sur l'origine de la vie, le créationnisme, voulait alors que les êtres vivants soient des créations immuables de Dieu. Bien que l'homme ait pratiqué la sélection des individus dans l'élevage des animaux depuis la préhistoire, ce n'est qu'au XVIII^{e} siècle qu'Erasmus Darwin dans « Zoonomie, ou lois de la vie organique » [Dar96], énonce la première théorie de l'évolution, le transformisme. Cette idée est alors formalisée par Jean-Baptiste Lamarck : le lamarckisme accorde aux êtres vivants la capacité d'évoluer et de transmettre leurs évolutions de part une tendance interne à la complexification contrée par les circonstances extérieures.

Au XIX^{e} siècle, Charles Darwin s'oppose au lamarkisme dans « De l'origine des espèces par voie de sélection naturelle » [Dar59], en émettant l'idée de la sélection du plus apte parmi des individus naturellement variants. Le darwinisme était néanmoins incapable d'expliquer les mécanismes régissant l'apparition et la transmission des changements d'une génération à l'autre. Il faudra attendre le XX^{e} siècle pour que le généticien Gregor Mendel explique ces mécanismes par la fréquence d'apparition des gènes au sein d'une population. C'est finalement la théorie synthétique de l'évolution biologique ou néodarwinisme, intégrant les théories de Darwin, Mendel et de la génétique des populations qui est aujourd'hui communément admise par les biologistes. Ainsi, l'évolution n'est plus envisagée comme la transformation d'individus mais comme celle de populations. Le principe de base restant inchangé : la sélection naturelle.

Le parallèle entre évolution biologique et informatique n'est pas totalement fortuit. En effet, l'évolution des systèmes informatiques, tout comme l'évolution biologique, se manifeste par une accumulation de petites modifications. De plus, selon Lehman [BL76], un système informatique utilisé dans un environnement réel doit nécessairement évoluer, faute de quoi, il devient de moins en moins utile dans cet environnement et tend à disparaître.

Dans un contexte où les technologies de communication et le multimédia évoluent à grande vitesse, et où une part importante des systèmes informatiques ne remplissent pas leurs cahiers des charges lors de leurs livraisons, il est nécessaire de concevoir

[1] théorie selon laquelle les êtres vivants restent semblables au cours du temps

des systèmes évolutifs sous peine de les voir rapidement devenir obsolètes. Malheureusement, la poursuite effrénée des nouvelles fonctionnalités est souvent engagée au détriment d'une solution plus adaptable et extensible, et donc plus pérenne.

La réalisation de systèmes évolutifs est depuis longtemps une question centrale en informatique. La diffusion importante de l'informatique auprès du grand public exacerbe cette problématique. Par exemple, on constate qu'une distribution Linux est avant tout choisie pour son système de diffusion et d'installation, et la fréquence de ses mises à jour. Néanmoins, de nombreux développeurs négligent l'importance de cette problématique et se contentent, comme seul moyen d'adapter et d'étendre les systèmes, de diffuser des versions modifiées des logiciels. Les utilisateurs sont alors en charge de l'application des mises à jour, ce qui impose l'arrêt et le redémarrage des systèmes en cours d'exécution. Ce vecteur de propagation des évolutions n'est pas simplement un désagrément pour les utilisateurs, mais trahit l'absence de prise en compte de la problématique de l'évolution des systèmes. En plus d'imposer des contraintes à l'utilisateur, la négligence coupable des développeurs contribue à la prolifération des virus informatiques. En effet, contrairement aux développeurs, les pirates informatiques ont eux bien compris les enjeux posés par l'adaptabilité des systèmes : ils propagent régulièrement des virus exploitant des failles de sécurité connues sachant pertinemment que le délai entre la découverte d'une faille et sa correction éventuelle sur le parc informatique mondial leurs laisse tout loisir d'opérer.

Contrairement à ce que pourrait laisser penser cet état de fait, la conception de systèmes extensibles a fait l'objet d'importantes recherches ayant abouties à des solutions originales. Néanmoins, ces approches sont difficilement applicables aux applications système patrimoniales : la qualité du code de ces systèmes est l'aboutissement de nombreux remaniements et corrections et ces systèmes sont soumis à de fortes contraintes de performance. Leur *refactoring* implique une dégradation de leur qualité et de leur performance pour un résultat dont l'adaptabilité reste à démontrer. En effet, il est difficile voire impossible d'anticiper quelles seront les évolutions futures, les interfaces permettant l'extensibilité d'un système se révèlent souvent inadaptées si ce n'est inutiles face aux réels besoins d'évolution.

À l'inverse, les travaux sur l'adaptation par la transformation à la volée des applications se confrontent mieux aux contraintes de performance des applications systèmes patrimoniales. En n'imposant pas de *refactoring* statique, ces approches maintiennent la qualité des systèmes, tout en permettant l'adaptation fine et performante des applications. Ces travaux se limitent néanmoins à des outils de manipulation de code à la volée, dès lors, ils demeurent réservés à une utilisation experte. Par manque de support langage, les adaptations par transformation de programme à la volée sont souvent complexes à concevoir et difficilement réutilisables.

1.1 Objectifs

Dans cette thèse, nous nous sommes fixés comme objectif de réconcilier adaptabilité dynamique et performance des systèmes informatiques. Pour cela, nous avons choisi de nous affranchir des approches *à priori*, souvent mises en défaut dans leur capacité à anticiper les besoins futurs notamment pour la correction de trous de sécurité. Aussi, nous nous sommes orientés vers une approche par réécriture à la volée du code binaire

exécuté par le processeur. En effet, l'application d'une mise à jour par une modification fine à la volée d'un système apparaît plus prometteuse pour les performances.

Néanmoins, la manipulation de code exécutable n'est pas chose triviale. En effet, pour la majorité des utilisateurs de langages de haut niveau, l'assembleur demeure obscur et non structuré. Aussi, pour réconcilier performance et aisance d'utilisation, nous avons choisi d'utiliser une approche langage : la programmation par aspects. Les aspects permettent d'exprimer du code à exécuter en réaction à l'exécution d'un code de base et ceci dans un langage proche de celui utilisé par ce dernier. Ce paradigme apparaît comme idéal pour l'expression d'adaptations et d'extensions de logiciels.

1.2 Contributions

Ce document décrit notre apport au domaine de l'évolution des systèmes informatiques patrimoniaux. Nos contributions portent principalement sur trois points : les langages d'aspects, les techniques de tissage d'aspects et d'injection de code à la volée, et les applications de la programmation par aspects.

1.2.1 Langage d'aspect

Nous avons conçu un langage d'aspect spécifiquement pour l'adaptation des systèmes C patrimoniaux. Notre langage définit des expressions rationnelles sur des événements du programme de base qui déterminent le déclenchement d'actions. L'expression d'adaptations par ce langage se fait donc en deux parties : en décrivant le code de l'adaptation (action) puis en exprimant où et quand dans l'exécution du programme à adapter, cette action doit être déclenchée. Afin que le développeur d'adaptation raisonne au niveau sémantique du programme à adapter, notre langage utilise des concepts et une syntaxe proches du langage C. Ainsi, l'exécution du code de l'adaptation peut être déclenchée lors d'appels à des fonctions ou lors de l'accès à des variables. En termes de contributions, notre langage d'aspect intègre la notion de protocole : il permet d'exprimer des relations séquentielles ou imbriquées entre plusieurs événements (appels de fonctions et/ou accès à des variables) [1, 3, 10]. L'expression d'enchaînements d'événements permet notamment de capturer les relations entre un programme et une bibliothèque. Notre langage autorise également l'expression d'adaptations réagissant aux interactions entre plusieurs applications et avec le système d'exploitation : une adaptation peut par exemple, être déclenchée dans le noyau Linux en réaction à l'exécution d'un appel système après l'exécution d'une fonction donnée dans une application [4].

1.2.2 Arachne

Pour les besoins de cette thèse, nous avons développé le prototype Arachne. Arachne est un système de programmation par aspects. Il comprend un compilateur pour notre langage d'aspect et un tisseur d'aspect. Le tisseur d'aspect d'Arachne fonctionne par réécriture à la volée du code binaire exécuté par le processeur et ce sans interruption de l'exécution. Notre tisseur fonctionne sur environnement Unix pour les plateformes Intel 32 bits. La technologie de tissage utilisée par Arachne n'est pas dépendante d'un compilateur ou d'un environnement d'exécution particulier et donc fonctionne sur des

applications patrimoniales [1, 3]. Arachne permet indistinctement le tissage d'aspects sur des applications en espace utilisateur et sur le noyau Linux [4].

En plus d'utiliser un mécanisme générique de réécriture de code, Arachne met à profit la sémantique du code exécutable pour fournir des mécanismes spécifiques de réécriture. L'utilisation de ces techniques permet de déclencher l'exécution des aspects nettement plus rapidement en comparaison aux autres tisseurs dynamiques d'aspects pour C [1, 2].

1.2.3 Applications

Nous avons voulu démontrer que le tissage d'aspect par réécriture de code à la volée est une solution viable pour l'évolution des systèmes patrimoniaux. Pour ce faire, il nous fallait établir que notre langage d'aspect est assez expressif pour concevoir des évolutions logicielles et que l'utilisation de la réécriture de code à la volée permet l'application des évolutions de manière sûre et performante.

Nous avons donc montré que notre approche peut être utilisée pour le débogage [9] et le monitoring [4] d'applications hautement performantes comme le cache Web Squid. Nous avons également confirmé que notre langage d'adaptation permettait d'exprimer des extensions logicielles par l'intégration dans Squid d'une politique de préchargement de contenu [2]. Finalement, nous avons présenté comment notre prototype Arachne pouvait être utilisé pour la correction de trous de sécurité dans des logiciels système comme Squid ou wu-ftpd [5, 6, 7].

1.3 Organisation du document

Ce document est structuré en trois parties. La première introduit le contexte de l'étude. Nous y présentons un état de l'art de l'adaptabilité et de l'extensibilité dans les systèmes informatiques. Nous distinguons les solutions existantes par rapport au moment de l'évolution, c'est-à-dire : d'une part les approches anticipées faisant intervenir des patrons de conception ou des langages particuliers, et d'autre part les approches non-anticipées.

Ensuite, nous présentons la contribution de cette thèse. En premier lieu, nous décrivons le langage que nous avons conçu pour exprimer des adaptations dans des logiciels. Puis, nous présentons l'outil associé, Arachne, réalisant l'implantation à la volée des adaptations dans les systèmes sans prérequis sur ces derniers.

Dans la dernière partie, nous montrons à l'aide d'évaluations de performance et de scénarios d'utilisation variés que notre proposition permet un compromis entre adaptabilité et performance sans anticipation lors de la conception et de la mise en production des systèmes. Enfin, nous dressons le bilan de nos travaux et ébauchons quelques perspectives.

1.4 Diffusion scientifique

Les travaux présentés dans ce document ont fait l'objet de diverses publications. On les distingue en deux catégories :

- Les publications portant sur le langage d'aspect développé pour notre tisseur [1][3, 4][10]. Ces publications sont discutées principalement dans le chapitre 4.
- Les publications décrivant les expérimentations de la programmation par aspects dynamique pour divers domaines d'applications allant de la sécurité [5][6, 7] à l'évolution de logiciels [2][8, 11], en passant par le débogage [9]. Ces publications sont discutées principalement dans le chapitre 6.

Journaux Internationaux

[1] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with Arachne. *Lecture Notes in Computer Science*, 1(1) :174–213, 2006. Transaction on Aspect-Oriented Software Development.

[2] Marc Ségura-Devillechaise, Jean-Marc Menaud, Nicolas Loriant, Thomas Fritz, Rémi Douence, Mario Südholt, and Egon Wuchner. Dynamic adaptation of the Squid web cache with Arachne. *IEEE Software*, 23(1) :34–41, 2006. Special Issue on Aspect-Oriented Computing.

Conférences internationales avec comité de lecture

[3] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with Arachne. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, pages 27–38, Chicago, IL, USA, March 2005. ACM Press.

[4] Nicolas Loriant and Jean-Marc Menaud. Generalized dynamic probes for the Linux kernel and applications with Arachne. In *Proceedings of the 2007 IADIS Applied Computing International Conference (AC'O7)*, Salamanca, Spain, February 2007.

[5] Nicolas Loriant, Marc Ségura-Devillechaise, and Jean-Marc Menaud. Server protection through dynamic patching. In *Proceedings of the 11th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'05)*, pages 343–349, Changsha, Hunan, China, December 2005. IEEE Computer Society.

Conférences nationales avec comité de lecture

[6] Nicolas Loriant, Marc Ségura-Devillechaise, and Jean-Marc Menaud. Des correctifs de sécurité à la mise à jour - audit déploiement distribué et injection à chaud. In *(DECOR'04), 1ère conférence Francophone sur le déploiement et la (re)configuration de logiciels*, pages 65–76, Grenoble, France, October 2004.

[7] Nicolas Loriant, Marc Ségura-Devillechaise, and Jean-Marc Menaud. Un bac à sable juste à temps – correctifs ciblés et injection à chaud. In *(CFSE'05), 5ème conférence française sur les systèmes d'exploitation*, Le Croisic, France, March 2005.

Ateliers internationaux avec comité de lecture et publication des actes

[8] Fabien Hermenier, Nicolas Loriant, and Jean-Marc Menaud. Power management in grid computing with Xen. In *Proceedings of 2006 ISPA workshop on Xen in High-Performance Clusters and Grid Computing Environments (XHPC'06)*, Sorrento, Italy, December 2006.

[9] Nicolas Loriant and Jean-Marc Menaud. The case for execution replay using a virtual machine. In *Proceedings of 2006 WETICE workshop on Emerging Technologies for Next-generation GRID (ETNGRID'O6)*, Manchester, UK, June 2006.

[10] Nicolas Loriant, Marc Ségura-Devillechaise, Thomas Fritz, and Jean-Marc Menaud. A reflexive extension to Arachne's aspect language. In *Proceedings of 2006 AOSD workshop on Open and Dynamic Aspect Languages (ODAL'O6)*, Bonn, Germany, March 2006.

[11] Nicolas Loriant, Marc Ségura-Devillechaise, and Jean-Marc Menaud. Software security patches – audit, deployment and hot update. In *Proceedings of the 4th AOSD workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS'O5)*, pages 25–29, Chicago, IL, USA, March 2005.

Première partie

Contexte de l'Étude

Ceux qui se sont sagement limités à ce qui leur paraissait possible n'ont jamais avancé d'un seul pas.

Mikhaïl Bakounine

Chapitre 2

Contexte de l'étude

Où nous présentons un état de l'art de l'évolution dynamique des applications systèmes patrimoniales. Nous discutons des designs adaptables et des environnements d'exécution dédiés. Nous discutons également de la transformation de programmes et ce, à travers deux techniques : la réécriture dynamique de code exécutable et la programmation par aspects. Nous montrons les limites des approches existantes et en déduisons notre proposition.

Sommaire

2.1 Designs extensibles **10**
2.1.1 La paramétrisation 11
2.1.2 Le chargement dynamique – plugins 11
2.1.3 La spécialisation de programmes 12
2.1.4 Architectures extensibles 12
2.2 Environnements d'exécution pour logiciels extensibles **13**
2.2.1 Programmation par composants 13
2.2.2 Composition de méta-niveaux 14
2.2.3 Environnements d'exécution dédiés 15
2.2.4 L'interposition 16
2.3 Transformation de programmes **17**
2.3.1 Les systèmes de réécriture de code 18
2.3.2 La programmation par aspects 22
2.4 Récapitulatif **24**
2.5 Proposition – Programmation par aspects par réécriture de code . . **25**

La nécessité de concevoir des applications adaptables n'est plus à démontrer. Nous assistons aujourd'hui à l'émergence toujours plus rapide des nouvelles technologies. Plus que jamais, les créateurs des systèmes informatiques ont besoin de méthodes et d'outils afin de poursuivre la course effrénée à l'intégration de nouvelles fonctionnalités. L'émergence de nouvelles technologies et la diversification des plateformes numériques, téléphones portables, *home cinéma*, *etc* font de l'adaptabilité et de l'extensibilité, un enjeu de pérennité pour les systèmes informatiques.

De plus, les systèmes informatiques ont atteint une complexité telle qu'ils sont hautement interdépendants. Parce que les applications et le système d'exploitation sont de plus en plus interdépendants, il est nécessaire d'envisager une solution globale à la problématique de l'évolution des systèmes informatiques.

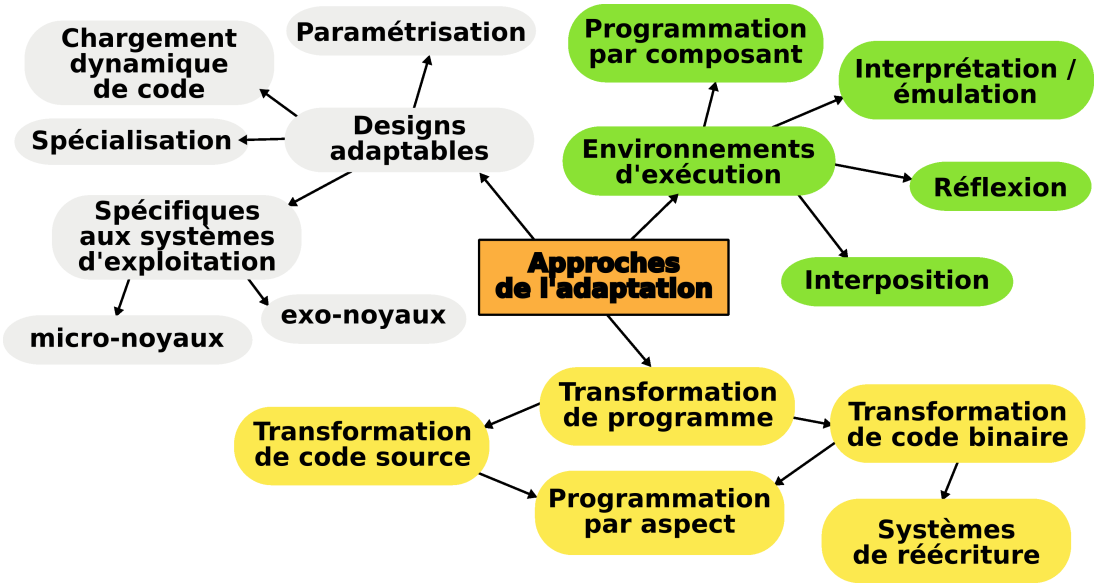


FIG. 2.1: Organisation de l'état de l'art

La construction d'applications et de systèmes d'exploitation adaptables a fait l'objet de nombreuses recherches originales. Dans cette partie, nous présentons un état de l'art des méthodologies et des outils pour l'adaptabilité des systèmes informatiques. Nous envisageons les recherches présentées par les mécanismes d'adaptation employés. L'organisation de la présentation de ces approches dans ce document est illustrée par la figure 2.1. Nous distinguons trois approches. D'abord, nous présentons les architectures adaptables et extensibles. Ce sont ces solutions qui sont les plus communément usitées. Ensuite, nous discutons des environnements d'exécution propices à l'adaptation logicielle. Enfin, nous terminons par les approches par transformation de code, parmi lesquelles, nous verrons notamment la programmation par aspects.

Pour chaque approche, nous évaluons dans quelle mesure celles-ci permettent l'adaptation des systèmes informatiques : permettent-elles l'adaptation, l'extension ? Nous voyons également si elles doivent être anticipées. Un autre critère d'évaluation porte sur le temps de l'adaptation : l'approche est-elle statique ou dynamique ? Enfin, nous voyons si cette approche est applicable aux systèmes patrimoniaux : l'approche nécessite-t-elle la modification de la chaîne de déploiement ? Finalement, pour chaque approche nous présentons des exemples d'implémentations à la fois pour les applications et surtout dans le cadre des systèmes d'exploitation. En effet, ces derniers sont des exemples typiques de systèmes patrimoniaux.

2.1 Designs extensibles

Dans cette partie, nous présentons les architectures logicielles extensibles. Ces solutions d'adaptation ont pour point commun d'être anticipées lors de la conception. Nous décrivons ici les solutions suivantes : la paramétrisation, le chargement dynamique de code et la spécialisation de programmes. Ensuite, nous présentons des approches architecturales de l'adaptation spécifiques aux systèmes d'exploitation.

2.1.1 La paramétrisation

La paramétrisation est une technique ancienne d'adaptation des logiciels. Elle consiste à modifier les valeurs de variables adaptant le comportement d'un logiciel. Par exemple, il est possible de modifier dynamiquement l'affichage des terminaux UNIX en modifiant les variables *row* et *column*. De même, il est possible dans le noyau Linux de modifier les paramètres de la pile IP/TCP. Ces paramètres contrôlent les fanions utilisés sur les paquets de contrôle ou encore les délais de réponse et/ou de retransmission des paquets. La paramétrisation des protocoles réseaux est une méthode utilisée par les administrateurs de systèmes informatiques pour contrer les prises d'empreintes[1] de systèmes d'exploitation [SMJ00]. Cette technique d'adaptation est largement utilisée. Néanmoins, elle est nécessairement anticipée lors de la conception et n'autorise que l'adaptation des fonctionnalités prédéfinies sans permettre l'intégration de nouvelles fonctionnalités.

2.1.2 Le chargement dynamique – plugins

Une approche de l'adaptation des logiciels est le chargement dynamique de code. Cette solution est mise en œuvre par le chargement, lors de l'exécution du logiciel, d'une extension : un «*plugin*». En pratique, l'extension ou l'adaptation est encapsulée dans une bibliothèque logicielle : une librairie partagée. Lors de l'exécution, le logiciel appelle explicitement le chargeur de code qui place en mémoire les données et le code exécutable de la librairie partagée. Ensuite, l'éditeur de liens réalise les branchements nécessaires à l'exécution entre le programme et le *plugin*, ou lorsque plusieurs bibliothèques sont utilisées sur une même interface, le programme peut appeler les fonctionnalités appropriées des bibliothèques suite à leur localisation par l'éditeur de liens. Aujourd'hui, le chargement dynamique de *plugins* est utilisé dans la majorité des applications bureautiques (lecteurs multimédia, clients de messageries instantanées, *etc*) et dans de nombreuses applications. Par exemple, le support du PHP (PHP : Hypertext Preprocessor), des bases de données ou des scripts dans le serveur Web Apache est réalisé par le mécanisme de chargement dynamique de code.

C'est également le chargement dynamique de code qui est utilisé dans les systèmes d'exploitation monolithiques modulaires. Par exemple, les modules du noyau Linux sont des instances des *plugins*. Les modules Linux peuvent être soit liés statiquement lors de la compilation du noyau soit chargés dynamiquement lors de l'exécution. Le noyau Linux possède son propre chargeur et son propre éditeur de liens pour le chargement dynamique de modules.

Les *plugins* permettent l'adaptation du logiciel : un logiciel peut charger et décharger des *plugins* réalisant différemment les mêmes fonctionnalités. Contrairement à la paramétrisation, les *plugins* réalisent également des extensions des fonctionnalités d'un programme. Cette approche reste limitée : les *plugins* doivent respecter une interface générique (initialisation/déchargement), les interfaces sur lesquelles ils viennent se fixer doivent être définies lors de la conception du logiciel et la composition des plugins doit être anticipée lors de la conception du système. Il est souvent difficile d'anticiper

[1] Technique qui consiste à observer le comportement de la pile TCP pour déterminer le système d'exploitation d'une machine distante.

les interfaces qui s'avéreront utiles ultérieurement : dans la pratique, les *plugins* sont souvent modularisés à posteriori.

La paramétrisation est donc une externalisation des données du logiciel, ces données peuvent alors être manipulées par un tiers. Les *plugins* sont eux une manière externaliser le code ou d'utiliser un code créé par un tiers.

2.1.3 La spécialisation de programmes

La spécialisation de programme est une technique d'optimisation qui exploite des informations connues pendant l'exécution d'un programme [Jon96, CHL+98a, CHL+98b]. La spécialisation est une transformation volatile d'un code pour des valeurs constantes sur un sous-ensemble des données de ce code. Elle génère pour les valeurs spécifiées, une version optimisée du code dont l'empreinte mémoire est souvent plus réduite. En ce sens, elle ne permet que d'optimiser des comportements prédéfinis, la spécialisation de programme se rapproche de la paramétrisation. Bien qu'elle puisse être appliquée à la volée, la spécialisation de programmes ne permet que des adaptations anticipées.

2.1.4 Architectures extensibles

Les noyaux monolithiques modulaires sont une manière d'instancier pour les systèmes d'exploitation, l'approche de l'adaptabilité par chargement dynamique de code. La recherche sur l'architecture des systèmes d'exploitation propose également deux autres architectures plus adaptables que les noyaux monolithiques : les micro-noyaux [BRS+85, HHL+97, RAA+88, MvRT+90] et les exo-noyaux [EKO95].

Les micro-noyaux noyaux sont une forme minimale de noyaux. La philosophie des micro-noyaux est de décomposer le noyau en un nucléus et un ensemble de serveurs. Le nucléus fournit les mécanismes minimaux à la mise en œuvre des serveurs : un gestionnaire de processus, un gestionnaire mémoire, un ordonnanceur, et un système de communication inter-processus. Les services offerts par le système d'exploitation sont implémentés dans les serveurs et accessibles par envois de messages.

Les micro-noyaux présentent l'avantage d'avoir une meilleure tolérance aux fautes : la séparation des fonctionnalités en serveurs permet d'isoler les différentes parties du code. Cette isolation est fournie soit par le matériel par l'utilisation des protections mémoire, soit par vérification de propriétés sur le logiciel. Par exemple, le micro-noyau SPIN [BSP+95] implémenté en MODULA-3 [CDJ+89] utilise les protections mémoire matérielles. SPIN propose d'exécuter les serveurs en collocation avec le nucléus (c'est-à-dire sans isolation par protection matérielle) lorsque le serveur est codé en MODULA-3. En effet, les propriétés du langage MODULA-3 permettent de garantir l'isolation des erreurs. Une isolation logicielle alternative est proposée par le micro-noyau VINO [SESS96]. VINO utilise également l'isolation mémoire fournie par le matériel mais impose également l'utilisation de requêtes transactionnelles entre les serveurs du noyau. Ceci permet d'éviter la propagation d'erreurs comme la non libération de ressources partagées. Camille-NG est un micro-noyau pour l'embarqué qui utilise également les vérifications de code et le chargement dynamique de code pour permettre l'adaptabilité dynamique [GHSR07].

L'adaptation des micro-noyaux est réalisée par remplacement des serveurs. Cette approche repousse les limites du noyau et permet donc d'adapter du code inadap-

table dans un noyau monolithique. Néanmoins, cette solution seule ne permet pas une adaptation fine.

Les exo-noyaux vont plus loin dans le concept des micro-noyaux. Contrairement aux micro-noyaux qui reposent sur un nucléus pour faire fonctionner les serveurs, l'approche des exo-noyaux est de ne fournir aucun service. En effet, un exo-noyau est un multiplexeur de ressources : il ne fait qu'allouer des ressources (disque, mémoire, *etc*) sans les gérer. Ainsi, chaque application peut implémenter ses propres gestionnaires de ressources (contrairement au micro-noyau où un ou plusieurs serveurs sont responsables des ressources). Cette architecture est particulièrement adaptée à la recherche de performances puisque les applications contrôlent de bout en bout les ressources qui leurs sont allouées. Les exo-noyaux repoussent totalement la limite du noyau. Il est donc possible bien que compliqué, de construire des applications totalement spécifiques à un contexte d'exécution. Néanmoins, les exo-noyaux ne fournissent pas de moyens spécifiques à l'adaptation à la volée mais garantissent l'adaptabilité par l'externalisation totale des fonctionnalités à la manière des *plugins*.

2.2 Environnements d'exécution pour logiciels extensibles

La conception d'environnements d'exécution dédiés est une approche permettant l'adaptation transparente de programme. Au lieu de modifier le comportement du programme, cette approche consiste à modifier l'interprétation de ce programme. Dans cette partie, nous présentons les différentes techniques de mise en œuvre des environnements d'exécution pour l'extension des logiciels.

2.2.1 Programmation par composants

La programmation par composants est une approche de génie logiciel qui consiste à architecturer un système en composants métiers ou logiques. Szyperski énonce les critères définissant un composant [Szy97]. Un composant est une unité autonome et ne dépend pas du contexte. Les composants fonctionnent comme des boîtes noires. Enfin, les composants sont composables et réutilisables. Par certains aspects, les librairies partagées Unix, les bibliothèques de classes Java sont des composants logiciels.

Les modèles de composants, Fractal [BCL+06], Java Beans [MH99], *etc* intègrent un environnement d'exécution, des couches logicielles pour la communication, un langage de spécification des interfaces des composants et un langage de description de l'organisation des composants : ADL (Architecture Description Languages). La vérification de l'équivalence entre deux contrats permet de remplacer un composant par un autre soit statiquement soit lors de l'exécution du système. De même, la reconfiguration à la volée de l'architecture des composants permet l'adaptation à la volée des systèmes [LBH04].

Les systèmes eCos [Mas02], tinyOS [Gro05] et Think [FSLM02] ont choisi la programmation par composants pour offrir une implémentation réutilisable et adaptable des noyaux de systèmes d'exploitation. Think est une implémentation en C du modèle de composants Fractal. Think fournit la librairie de composants KORTEX qui fournissent les fonctionnalités fréquemment utilisées dans les systèmes d'exploitation :

gestionnaires mémoires, ordonnanceurs, systèmes de fichiers, *etc.* L'utilisation de la programmation par composants permet ici d'obtenir un système d'exploitation personnalisé et adaptable. L'adaptabilité est obtenue par remplacement de composants et l'extensibilité par ajout.

La programmation par composants apporte un intérêt indéniable vis-à-vis des approches précédentes. Il est possible de vérifier par avance que les modifications apportées au système respectent les contrats établis par les interfaces des composants. Néanmoins, dans une approche «boîtes noires», les vérifications ne portent pas sur l'état des composants avant, pendant et après les reconfigurations. De plus, la décomposition initiale du système en composants limite la granularité des évolutions possibles, et se confronte à la problématique de la décomposition dominante [OT00]. La capacité des approches par composant à évoluer demeure limitée à des adaptations anticipées et marginalement non anticipées.

2.2.2 Composition de méta-niveaux

Une approche de l'adaptation est la composition de méta-niveaux. Les systèmes construits sur des méta-niveaux sont dits réflexifs. La réflexion est la capacité d'un programme à entretenir, consulter et potentiellement modifier une représentation de lui-même et de son exécution [Smi82]. La réflexion regroupe deux ensembles de techniques : les techniques d'introspection offrent les moyens au programme pour consulter sa représentation et son état, et les techniques d'intercession permettent de modifier cette représentation et cet état. Bien qu'indépendante d'un paradigme de programmation, la réflexion est principalement liée à la programmation orientée objets. Les langages implémentant la réflexion sont principalement orientés objet : SmallTalk [GR85, GK76], Java [GJSB05], C# [ECM02], Python [Ros03], Ruby, PHP, Objective-C. On distingue deux types de réflexion. D'une part, la réflexion structurelle qui concerne la représentation du programme et de son état. Elle concerne la représentation des données, des types et des traitements. Elle permet la manipulation du programme et sa modification. D'autre part, la réflexion comportementale qui concerne plus particulièrement l'exécution du programme et l'environnement d'exécution. La réflexion comportementale rend possible la modification de la manière dont est exécuté le programme afin de l'adapter au mieux à l'environnement. Les méta-objets sont une approche structurée de la réflexion [KRB91]. Un protocole de méta-objets représentant le programme et son état et son interpréteur par des objets à part entière au même titre que les objets métiers. Ces méta-objets étant eux-mêmes des objets, ils possèdent également des méta-méta-objets associés. Pour éviter un empilement infini des couches de méta-objets, les protocoles de méta-objets définissent un niveau de méta-objet terminal qui doit être auto-suffisant.

Un protocole de méta-objets spécifie la manière dont les objets et les méta-objets sont liés : les interfaces satisfaites par les méta-objets. Ce lien définit les instructions de l'interprétation du programme qui peuvent être redéfinies par les méta-objets. Le grain auquel il est possible d'adapter le programme est donc directement lié au détail du protocole de méta-objets. Comme les méta-objets sont eux-mêmes réifiés par des méta-méta-objets, il est possible de composer des modifications du programme par composition des méta-niveaux.

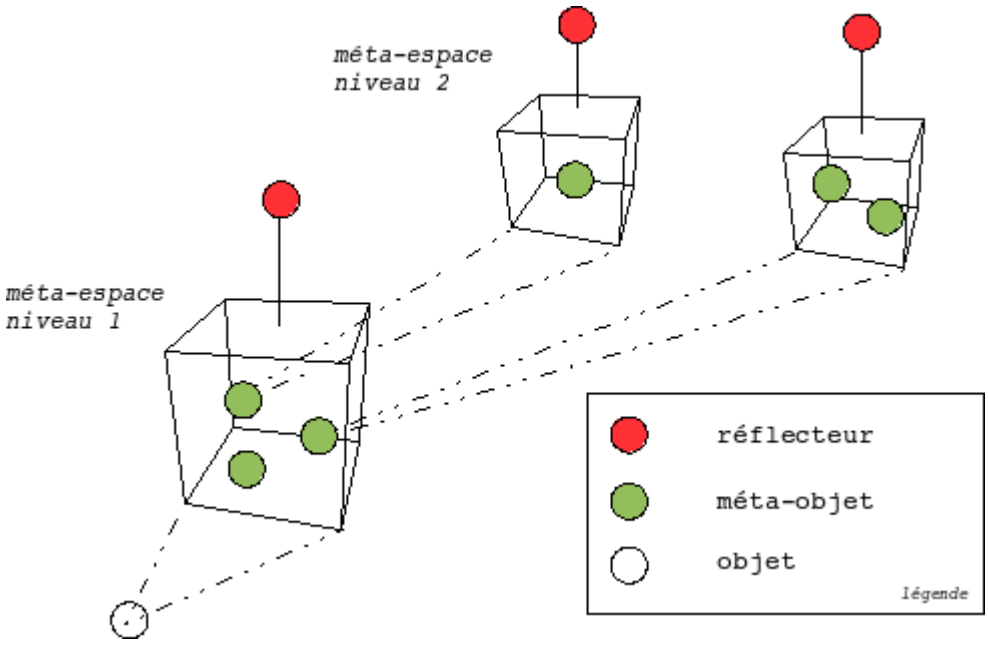


FIG. 2.2: Organisation du protocole de méta-objet dans Apertos

Apertos est un système d'exploitation adaptable utilisant la programmation orientée objet et la réflexion développé par Sony [Yok92]. Apertos implémente un protocole de méta-objets illustré par la figure 2.2. Les objets y sont regroupés en méta-espaces, un méta-niveau pouvant contenir un ou plusieurs méta-espaces et tout objet pouvant être associé à des méta-espaces de niveaux supérieurs. Les objets des niveaux bas implémentent les fonctionnalités métiers et les méta-niveaux fournissent des fonctionnalités support comme les traces ou la persistance. Les fonctionnalités support étant implémentées par manipulation de l'interprétation des objets métiers, elles sont facilement réutilisables. De plus, Apertos propose une interface appelée réflecteur regroupant les mécanismes facilitant la migration des objets et méta-objets entre les différents méta-niveaux. Ces mécanismes facilitent la composition des adaptations par composition des méta-niveaux.

Bien que l'utilisation de la réflexion permette l'adaptation et l'extension à la volée des logiciels, cette approche pose plusieurs problèmes. D'une part, la réification de l'environnement d'exécution d'un système est incompatible avec les performances recherchées par les applications dites «systèmes». D'autre part, l'adaptation par modification de l'interprétation du programme n'est pas intuitive. Il est notamment difficile de garantir que la modification de l'interprétation du programme soit toujours capable d'exécuter le programme [BSLR98].

2.2.3 Environnements d'exécution dédiés

Cette approche consiste à exécuter un programme dans un environnement d'exécution spécifique, par exemple, un «interpréteur» de code. L'interprète analyse une instruction du programme, l'exécute puis passe à la suivante. L'interprétation de programme augmente la portabilité des programmes et facilite également l'implémentation

et le débogage : pas de compilation et donc pas de perte d'information entre le code source et le code compilé. Aujourd'hui, la majorité des langages interprétés s'exécutent sur machines virtuelles, celles-ci se placent à mi-chemin entre interpréteurs et compilateurs.

L'interprétation de programme n'est pas limitée aux langages interprétés comme Java, Ruby, Python ou PHP. Par exemple, CINT est un interpréteur de code source C [DG87]. Cet exemple se rapproche plus de l'émulation de code. Les émulateurs comme QEMU [Bel05], s'orientent plus vers la simulation d'un matériel, par exemple, pour offrir la compatibilité d'un programme ancien sur une architecture moderne.

L'adaptation par l'utilisation d'environnements d'exécution dédiés consiste donc à modifier l'interprète ou l'émulateur du programme. Cette solution se rapproche de la réflexion bien que l'adaptation ne soit pas faite par le programme lui-même mais par la modification de l'interpréteur. Par cette similitude, les environnements d'exécution dédiés présentent les mêmes limitations pour l'évolution des systèmes informatiques.

2.2.4 L'interposition

L'interposition est une autre approche de l'adaptabilité des systèmes informatiques. Le principe de base de l'interposition est de placer un code adaptant ou étendant le comportement d'un système entre deux entités de ce système. Autrement dit, l'interposition consiste à capturer des événements traversant une interface pour déclencher une action. Le traitement des événements interceptés détermine le type d'évolution : soit l'action vient modifier le comportement d'origine et l'événement est étoufé, soit l'action étend ce comportement et l'événement est retransmis. Les entités entre lesquelles sont greffées les adaptations dépendent principalement du paradigme de programmation utilisé lors de la conception du système : programmes, composants, librairies partagées, *etc.*

Le mécanisme d'interposition peut être utilisé pour l'adaptabilité des systèmes d'exploitation. SLIC est un prototype pour l'adaptation du noyau Solaris [GPRA98]. SLIC propose des *dispatchers* qui permettent d'intercepter les points d'entrées entre les applications utilisateur et le noyau solaris : les appels systèmes et les interruptions. Ces dispatchers sont implémentés par réécriture des vecteurs d'interruptions. Les extensions sont codées en C. SLIC expose aux développeurs d'extensions une interface permettant comme le montre la figure 2.3, d'appeler explicitement le comportement d'origine du système ou de déclencher les extensions. SLIC offre la possibilité de charger les extensions soit en espace utilisateur, soit dans l'espace noyau. L'espace noyau permet aux extensions d'utiliser les bibliothèques standards de développement. Tandis que le chargement en espace noyau présente de meilleures performances : l'interception des appels systèmes se faisant en espace noyau, on évite ainsi deux changements de contexte supplémentaires comme le montre les résultats du tableau 2.4. Ce point est d'autant plus critique que les routines de gestion d'interruptions sous Solaris ne sont pas réentrantes.

Le projet FUSE (Filesystem in Userspace) est une autre solution d'interposition : FUSE exporte le gestionnaire de systèmes de fichiers du noyau Linux en espace utilisateur [teac]. FUSE permet ainsi de concevoir des systèmes de fichiers en espace mémoire utilisateur. Le project BOSSA est également une solution d'interposition pour l'adaptation de la politique d'ordonnancement du noyau Linux [MLD05].

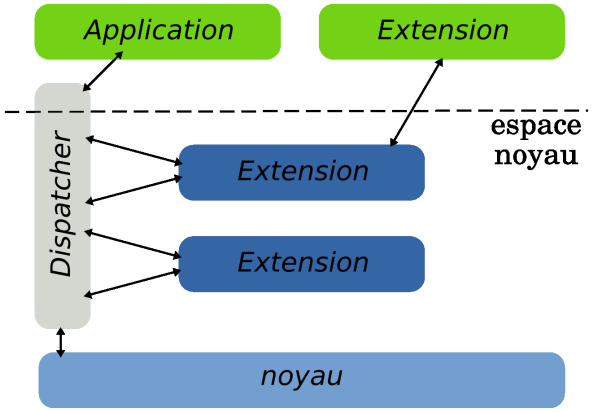


FIG. 2.3: Adaptation par interposition avec SLIC

	getpid()	
	Time (μs)	delta
Kernel non modifié	2.82	-
dispatcher sans extension	3.16	0.34
dispatcher avec extension vide (en espace noyau)	4.38	1.22
dispatcher avec extension vide (en espace utilisateur)	61.83	57.47

FIG. 2.4: SLIC : surcoût de l'interposition sur plateforme ULTRASPARC, SOLARIS 2.5.1, moyennes sur 10000 exécutions

Le principe de l'interposition présente plusieurs avantages. L'interposition ne nécessite pas de modification du programme de base. De plus, cette approche est transparente pour le programme de base et il est possible de composer explicitement les actions ajoutées. Néanmoins, l'adaptabilité rendue possible par interposition est limitée par les choix de conception du programme de base : les adaptations ne peuvent être greffées que sur les interfaces «interposables» du programme. L'interposition est d'une manière, l'application des *plugins* à du code patrimonial. La multiplication des interfaces dans un système nuisant à sa compréhension, inévitablement, la découpe logique du système ne pourra satisfaire tous les besoins d'adaptation [OT00].

2.3 Transformation de programmes

Les approches examinées jusqu'à présent restent limitées à des adaptations anticipées lors de la conception du système ou impliquent un surcoût important lors de l'exécution de part la modification de l'environnement d'exécution. Nous présentons ici les techniques d'adaptation par transformation de programme. Ces techniques n'imposent pas un surcoût à l'exécution car elles ne reposent pas sur un environnement d'exécution spécifique ou sur un modèle de conception. De plus, elles repoussent le moment de l'adaptation dans le cas de la transformation de programme à la volée. Nous présentons ici deux mécanismes de transformation de programme. D'abord, nous décrivons les techniques de réécriture des programmes exécutable, puis nous introduirons la programmation par aspects.

2.3.1 Les systèmes de réécriture de code

L'instrumentation de code à la volée trouve ses origines dans les débogueurs des plateformes DEC des années 60. La technique du «code splicing» consiste alors à écraser une instruction par une instruction de saut afin de détourner l'exécution d'un programme vers le débogueur. Cette technique permet de modifier n'importe quelle instruction d'un programme sans préparation particulière. Elle a été reprise par tous les systèmes de réécriture de code dans des buts divers : l'analyse et le débogage [PKFH02, MR07], la mesure de performance [Kue95, Moo00, TM99b], ou l'optimisation [BDB00, PAB+95].

Vulcan est un système de réécriture de code exécutable à la volée [ESV01] développé par Microsoft Research pour adresser les problèmes d'adaptabilité dans les applications distribuées. Vulcan permet la réécriture de code exécutable pour les applications Win32 sur plateformes Intel 32 bits. Vulcan permet aux développeurs de concevoir un programme C++ qui modifie le code exécutable d'une application. Vulcan offre une interface C++ pour la manipulation d'une représentation abstraite du code exécutable. Vulcan commence par analyser le code exécutable de l'application et crée une représentation abstraite à base de routines, de blocs et d'instructions. C'est cette représentation qui est manipulée par le programmeur d'adaptation pour modifier le programme exécutable. Lors de l'application de l'adaptation Vulcan stoppe l'exécution du programme et modifie bloc par bloc le programme avant de le redémarrer. Vulcan permet de littéralement écrire un programme qui va en modifier un autre. Par exemple, le listing 2.1 réalise l'optimisation des sauts inconditionnels successifs d'un programme en les remplaçant par un saut direct vers la destination finale de la chaîne. Vulcan est un outil extrêmement puissant dont l'utilisation est très compliquée. En effet, le programmeur d'adaptation doit écrire un programme qui manipule les abstractions du code exécutable (instructions, blocs, routines) qui n'ont que peu de rapport avec le code source du programme. De plus, Vulcan ne s'occupe que d'appliquer les modifications décrites par le programmeur, c'est ce dernier qui doit s'assurer que l'adaptation ne va pas violer la cohérence du code exécutable.

Kerninst [TM99b, TM99a, Tam01] est un outil de réécriture dynamique de code pour le noyau Solaris sur architecture SPARC. Contrairement à Vulcan, Kerninst est totalement dynamique : c'est-à-dire que la réécriture de code ne nécessite pas de stopper l'exécution du noyau. Kerninst utilise la technique du «code splicing» pour injecter du code dans des noyaux non-modifiés. Tout comme Vulcan, Kerninst propose une API C++ pour la modification du code du noyau. Le développeur d'adaptation construit donc un programme qui va instrumenter le code binaire du noyau. Kerninst fournit également une API pour générer du code binaire. Par exemple, l'expression $A = (B * C + D)/(E - F)$ est construite par le code du listing 2.2 où l'on suppose que les variables A, B, C, D, E et F ont été allouées précédemment. Le code généré est ensuite automatiquement préfixé et suffixé par des routines de sauvegarde et de restauration des registres utilisés par le noyau à l'endroit de l'injection du code. Cette approche améliore celle de Vulcan puisqu'ici le programmeur d'adaptation ne se soucie pas de l'état (registres utilisés) du programme. L'utilisation de Kerninst demeure néanmoins compliquée et inadaptée à des évolutions significatives comme l'atteste l'exemple du listing 2.3. Cette exemple permet l'insertion d'un compteur d'appels sur une fonction du noyau dont le nom est passé en argument. (Cet exemple n'a pas d'intérêt particulier

```
// "BranchChain" realise l'optimisation d'une succession
// de sauts inconditionnels
void BranchChain (VInst * pInst) {
    for (;; pInst = pInstTarget) {
        VBlock * pBlockTarget = pInst->BlockTarget();
        VInst * pInstTarget = pBlockTarget->FirstInst();

        if (COp::IsUnCondBranch(pInstTarget->Opcode()))
            pInst->SetBlockTarget(pInst->BlockTarget());
        else break;
}}

// "Peephole" execute la fonction BranchChain sur tous
// saut inconditionnel du code binaire
void Peephole (VProg * pProg) {
    for (VComp * pComp = pProg->FirstComp(); pComp;
        pComp->Next()) {
        for (VProc * pProc = pComp->FirstProc();
            pProc; pProc->Next()) {
            for (VBlock * pBlock = pProc->FirstBlock();
                pBlock; pBlock->Next()) {
                for (VInst * pInst = pBlock->FirstInst();
                    pInst; pInst->Next()) {
                    if (COp::IsUnCondBranch(pInst.Opcode()))
                        BranchChain(pInst);
        }}}
        pComp->Write();
}}

// "main" execute la fonction Peephole sur le programme
// ou process passe en argumentz
int main (int argc, char ** argv) {
    VProg * pProg;

    if (IsProcessId(argv[1]))
        pProg = VProg::Open(GetProcessId(argv[1]);
    else pProg = VProg::Open(argv[1]);

    Peephole(pProg);
    return 0;
}
```

Listing 2.1: Optimisation des branchements inconditionnels consécutifs avec Vulcan

```
/*
    'A = ((B * C) + D) / (E - F)'
*/

kapi_arith_expr assign_A(
    kapi_assign, A,
        kapi_arith_expr(kapi_divide,
            kapi_arith_expr(kapi_plus,
                kapi_arith_expr(kapi_times, B, C), D),
                    kapi_arith_expr(kapi_minus, E, F)));
```

Listing 2.2: Génération de l'expression $A = (B * C + D)/(E - F)$ par l'API Kerninst

à part celui d'illustrer la complexité d'utilisation de Kerninst).

```
// count_func.C - mutator to count the number of entries to a user-specified
//                kernel function

#include <iostream>
#include <time.h>
#include <stdlib.h>
#include <unistd.h>
#include "kapi.h"

using namespace std;

// Correct usage is: "count_func hostname port# module function"
// "hostname" - the name of the host where kerninstd is running
// "port#" - the listening port supplied by kerninstd
// "module" - the name of a kernel module
// "function" - the name of a function of interest within module
int usage(char *arg0) {
    cerr << "Usage: " << arg0 << " <hostname> <port#> <module> <function>\n";
    return (-1);
}

// The global kapi_manager object, starting point for all KAPI interactions
kapi_manager kmgr;

void error_exit(int retval) {
    kmgr.detach();
    exit(retval);
}

int main(int argc, char **argv) {
    int rv = 0;

    if (argc != 5) {
    return usage(argv[0]);
    }

    // Grab the arguments
    const char *host = argv[1];
    int port = atoi(argv[2]);
    const char *modName = argv[3];
    const char *funcName = argv[4];

    // Attach to kerninstd on the target host using the supplied port
    if ((rv = kmgr.attach(host, port)) < 0) {
        cerr << "attach error\n";
        return rv;
    }

    // Find the module of interest, initialize kmod
    kapi_module kmod;
    if ((rv = kmgr.findModule(modName, &kmod)) < 0) {
        cerr << "findModule error\n";
        error_exit(rv);
    }

    // Find the function of interest, initialize kfunc
    kapi_function kfunc;
    if ((rv = kmod.findFunction(funcName, &kfunc)) < 0) {
        cerr << "findFunction error\n";
        error_exit(rv);
    }

    // Find the instrumentation point: the entry to the function
    kapi_vector entries;
    if ((rv = kfunc.findEntryPoint(&entries)) < 0) {
        cerr << "findEntryPoint error\n";
        error_exit(rv);
    }

    // Allocate space for the counter in the kernel space. The kptr_t
    // type denotes a pointer to kernel memory, but is defined in
    // kapi.h as an unsigned integer of the appropriate size
    kptr_t addr;
    unsigned size = sizeof(kapi_int_t);
    if ((addr = kmgr.malloc(size)) == 0) {
        cerr << "malloc error\n";
```

```
        error_exit(addr);
    }

    // Initialize the kernel memory, by copying from local variable
    kapi_int_t user_copy = 0;
    if ((rv = kmgr.memcopy(&user_copy, addr, size)) < 0) {
        cerr << "memcopy error\n";
        error_exit(rv);
    }

    // Create an int variable at the allocated space
    kapi_int_variable intCounter(addr);

    // Generate code to increment the variable atomically. The use of
    // kapi_atomic_assign is suggested versus kapi_assign when the host on
    // which kerninstd is running is a multi-processor, since an atomic
    // assignment prevents race conditions due to simultaneous access to
    // the intCounter variable
    kapi_arith_expr code(kapi_atomic_assign, intCounter,
             kapi_arith_expr(kapi_plus, intCounter,
                     kapi_const_expr(1)));

    // Insert the snippet code at the function entry point, and record
    // the value of the handle returned for later use in removal. Note that
    // the following code only inserts the snippet at the first entry
    // point, while it is possible that a kernel function may have
    // multiple entry points due to interprocedural jumps into the middle
    // of the function. In such a case, the insertSnippet call should
    // be made for each element in the entries kapi_vector.
    int sh;
    if ((sh = kmgr.insertSnippet(code, entries[0])) < 0) {
        cerr << "insertSnippet error\n";
        error_exit(sh);
    }

    // Loop for twenty seconds
    time_t t = time(NULL);
    while (time(NULL) - t < 20) {
        // Handle any pending KAPI events
        kmgr.handleEvents();

        // Read the in-kernel value of intCounter
        if ((rv = kmgr.memcopy(addr, &user_copy, size)) < 0) {
            cerr << "memcopy error\n";
            error_exit(rv);
        }
        // The client's byte ordering may not match that of the kernel
        kmgr.to_client_byte_order(&user_copy, size);
        cout << "# Entries = " << user_copy << endl;

        // Sleep for one second before continuing loop
        sleep(1);
    }

    // Remove the inserted code snippet from the function entry point
    // using the snippet handle returned by the call to insertSnippet
    if ((rv = kmgr.removeSnippet(sh)) < 0) {
        cerr << "removeSnippet error\n";
        error_exit(rv);
    }

    // Free the kernel memory allocated to hold intCounter
    if ((rv = kmgr.free(addr)) < 0) {
        cerr << "free error\n";
        error_exit(rv);
    }

    // Detach from the kerninstd
    if ((rv = kmgr.detach()) < 0) {
        cerr << "detach error\n";
    }

    return rv;
}
```

Listing 2.3: Insertion d'un compteur d'appels de fonction avec Kerninst

SystemTap [PCE+05] est un projet libre qui étend le projet KProbes [Moo01]. C'est

```
#!/usr/bin/env stap
global syscalls

function print_top () {
    cnt=0
    log ("SYSCALL\t\t\t\tCOUNT")
    foreach ([name] in syscalls-) {
        printf("%-20s\t\t%5d\n",name, syscalls[name])
        if (cnt++ == 20)
            break
    }
    delete syscalls
}

probe kernel.function("sys_*") {
    syscalls[probefunc()]++
}

probe timer.ms(5000) {
    print_top ()
}
```

Listing 2.4: Un script SystemTap pour l'affichage des 20 appels systèmes les plus utilisés sur les 5 dernières secondes

également un outil d'instrumentation de code à la volée. Il fonctionne pour les noyaux Linux sur les architectures Intel. SystemTap se distingue de Vulcan et Kerninst en facilitant le travail des développeurs. En effet, contrairement à ces deux outils, il fournit un langage de script pour injecter du code dans le noyau. C'est-à-dire que le développeur ne va pas écrire un programme pour adapter le noyau mais uniquement l'adaptation. Pour cela, SystemTap propose un langage de description de sonde. Ce langage permet de sélectionner les zones du code où injecter les adaptations. Contrairement aux outils précédents, l'injection ne se fait pas par rapport à des adresses dans le code binaire, mais par rapport à des noms de fonctions ou des lignes dans les sources du noyau. SystemTap se charge ensuite de résoudre ces points d'injection grâce aux informations de débogage du noyau. De plus, contrairement à Vulcan ou Kerninst, les adaptations ne sont pas programmées directement en assembleur mais dans un langage de script mêlant AWK et un sous-ensemble du langage C (l'exemple du listing 2.4 permet d'afficher toutes les 5 secondes les 20 appels systèmes les plus utilisés). L'utilisation de SystemTap est nettement plus simple que celle des outils précédents puisque le développeur raisonne au niveau des sources plutôt qu'au niveau de l'assembleur. Néanmoins, l'utilisation de SystemTap reste limitée au monitoring et à des adaptations simples : il ne permet pas de remplacer des portions de code (uniquement des sondes).

Les systèmes de réécriture de code sont des outils puissants. Ils permettent la modification de code au niveau assembleur. Par l'injection de code, ils fournissent les mécanismes pour l'adaptation et l'extension des systèmes à la volée et sans préparation. Leur utilisation demeure néanmoins réservée à des experts du code binaire. En effet, la cohérence des modifications apportées à un système en cours d'exécution est laissée à la charge du développeur d'adaptation.

2.3.2 La programmation par aspects

La programmation par aspects est une approche langage de la transformation de programmes. La programmation par aspects vise à répondre aux problèmes posés par la tyrannie de la décomposition dominante : l'architecture d'un logiciel est souvent influencée par les objectifs métiers, dans ce cadre, les fonctionnalités annexes (traces,

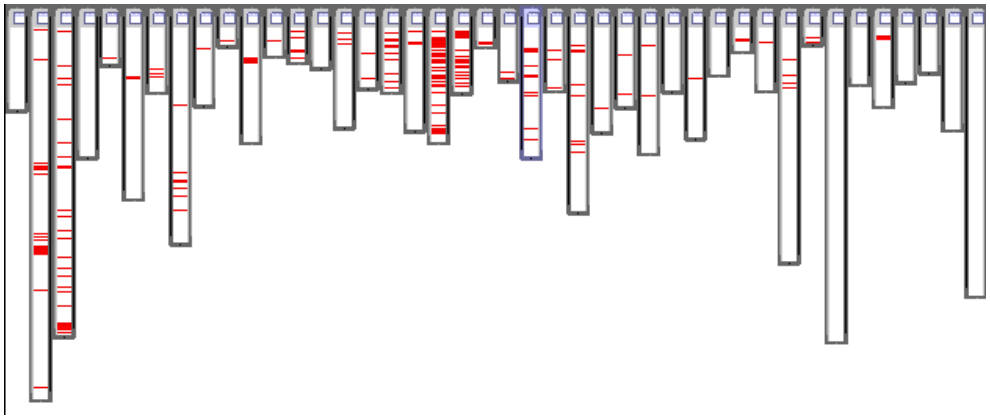


FIG. 2.5: La préoccupation *logging* dans le code source du module Tomcat d'Apache (Copyrigth® Gregor Kiczales)

authentification, journalisation, *etc*) s'inscrivent difficilement dans cette décomposition [OT00]. Dans la pratique, l'implémentation des fonctionnalités annexes est répandue sur l'ensemble du code de l'application. La figure 2.5 illustre la contamination de l'implémentation des journaux d'événements dans le code source de Tomcat. Cet enchevêtrement de l'implémentation des fonctionnalités support avec le code métier pose plusieurs problèmes. D'une part, l'implémentation des fonctionnalités annexes est difficile à comprendre car répandue sur l'ensemble de l'application. D'autre part, le code métier est obscurci par le code des fonctionnalités support. Enfin, le code est difficilement réutilisable car mal modularisé.

La programmation par aspects est un paradigme de programmation qui cherche à éviter l'enchevêtrement du code métier avec les fonctionnalités transverses [KLM+97]. La programmation par aspects propose un ensemble de langages et d'outils pour mieux modulariser l'implémentation des fonctionnalités transverses. Ce paradigme de programmation est indépendant des paradigmes classiques (procédurale, objet, fonctionnel, *etc*). Un système de programmation par aspects fournit un tisseur qui se charge d'ajouter une fonctionnalité implémentée dans un aspect, à un programme de base. Un aspect est défini par une «coupe» et une «action». L'action contient l'implémentation d'une fonctionnalité dans un langage de programmation classique et la coupe décrit les points dans le code du programme de base où sont tissées les actions. Ces points appelés «points de jonction» ne sont pas listés de manière exhaustive dans la coupe ; ce qui reviendrait à déplacer le problème de modularisation des fonctionnalités transverses. La coupe décrit par des expressions rationnelles l'ensemble des points de jonction. La coupe spécifie donc un contexte d'exécution qui, lorsqu'il est vérifié, doit déclencher l'action associée à la coupe. Par exemple, un aspect peut déclarer une action qui journalise l'authentification d'un utilisateur à chaque fois qu'une fonction *user_authentification* retourne la valeur *OK*. Le tissage va donc consister à injecter l'action à chaque retour de la fonction *user_authentification* lorsque la valeur de retour est égale à *OK*.

Le tissage des aspects consiste donc à mêler le code des aspects avec le code du programme de base. Le tissage d'aspect est soit statique soit dynamique. Un tisseur sta-

tique transforme le code source du programme de base pour y injecter les aspects. C'est le cas du tisseur d'aspect pour le langage Java, AspectJ [KLM+97]. JAsCo [VSCF05], JAC [PSD+04, PDFS01, PSDF01], et PROSE [PAG03] proposent quant à eux des tisseurs dynamiques. JAC introduit dans la hiérarchie de classes de Java, une chaîne d'objets agissant comme une couche d'interposition et permettant le tissage au runtime. Cette approche peut s'avérer coûteuse. JAsCo et PROSE modifient les mécanismes de chargement de classe de la machine virtuelle Java pour introduire le code des aspects. Le coût induit par cette approche est plus acceptable (8% sans aspect).

La programmation par aspects n'est pas limitée à un paradigme de programmation et Java n'en a pas l'exclusivité. Aspectual Caml est un système de programmation par aspects statique pour le langage Caml [MTY05]. AspectSharp [Teaa] permet la programmation par aspects pour dotNET. Le C et le C++ ne sont pas en reste bien que C4 [CKFS01, CKO+02] ne propose pas le tissage dynamique et qu'AspectC++ [LBS04, SPLS+06]) ne le permette qu'après la modification statique du programme à tisser. Les langages d'aspect de C4 et AspectC++ permettent de raisonner sur une large palette des concepts des langages C et C++ : appels de fonctions/méthodes, accès à des variables (de classe), interception des exceptions. Adams et De Schutter ont proposé un langage d'aspects raisonnant sur des continuations locales pour modulariser la gestion des valeurs de retours de fonction [AS07].

L'utilité de la programmation par aspects pour l'adaptation des systèmes d'exploitation n'est plus à démontrer [CKFS01, CKO+02]. Deux prototypes dédiés au tissages d'aspects à la volée dans le noyau Linux existent : TOSKANA [EF06] et KLASY [YKC06]. Le tissage à la volée présentant des difficultés techniques, le langage d'aspect de TOSKANA se limite à l'interception des appels de fonctions. Tandis que le langage d'aspect de KLASY permet l'interception des appels de fonctions, des accès à des variables et des accès à des champs de variables structurés, KLASY impose néanmoins l'utilisation d'un compilateur C modifié.

L'utilisation de la programmation par aspects présente un intérêt majeur pour l'adaptabilité des systèmes informatiques. Les aspects expriment des fonctionnalités exécutées en réaction à l'exécution d'un programme de base. Ce paradigme de conception est particulièrement adapté à l'expression d'adaptations ou d'extensions d'un logiciel. Malheureusement, il existe peu de systèmes de programmation par aspects à la volée pour C. Les systèmes existants sont soit limités dans leur pouvoir d'expression soit requièrent la modification de la chaîne de compilation. Ces systèmes sont alors peu adaptés à l'adaptation non anticipée des systèmes patrimoniaux.

2.4 Récapitulatif

Afin de récapituler cet état de l'art, le tableau 2.1 dresse le bilan des différentes approches présentées dans ce document. Pour chaque approche, le tableau indique leurs applications aux systèmes d'exploitation, si elles peuvent être appliquées dynamiquement, si elles permettent l'adaptation non anticipée et enfin si elles requièrent une modification de la chaîne de développement/compilation/déploiement.

Approche	Application aux SE	dynamique	non desgin-based	sans modification de la chaîne de compilation	support langage
paramétrisation	–	oui	non	oui	non
chargement dynamique de code	monolithiques modulaires	oui	non	oui	non
spécialisation de programme	–	oui	oui	non	non
micro/exo-noyau	–	–	non	oui	non
programmation par composants	Think, eCos	oui	non	non	non
réflexion	Apertos	oui	oui	non	non
intrepréteur–émulateur	–	oui	oui	non	non
interposition	Slic	oui	oui	oui	non
réécriture de code	Kerninst, SystemTap	oui	oui	oui	non
programmation par aspects	TOSKANA, KLASY	oui	oui	non	oui

TAB. 2.1: Récapitulatif par approche : applications aux systèmes d'exploitation, statiques, non-anticipée, sans modification de la chaîne de déploiment

2.5 Proposition – Programmation par aspects par réécriture de code

Les techniques d'adaptation existantes sont difficilement applicables aux systèmes informatiques patrimoniaux. En effet, toutes les solutions basées sur des designs de conception : paramètrisation, chargement dynamique, *etc*, supposent une prise en compte à la conception et présentent une capacité d'évolutivité limitée. De même les approches par environnements d'exécution dédiés impliquent des surcoûts à l'exécution incompatibles avec la recherche de performances imposées par les applications systèmes. Les techniques de réécriture de code exécutable par ailleurs offrent une grande capacité d'adaptation non-anticipée. En effet, ces approches permettent la modification fine et totale des systèmes. La réécriture de code est néanmoins extrêmement difficile à utiliser car elle requiert une maîtrise parfaite du système et du code exécutable de celui-ci.

L'adaptation par aspect est une approche puissante. Le tissage dynamique permet en effet de mettre en œuvre des adaptations non anticipées lors de l'implémentation du système de base. Cette approche retarde le moment de l'adaptation : il n'est pas nécessaire lors de la conception d'un logiciel de penser les interfaces qui seront potentiellement utiles à des adaptation futures ; C'est le langage de coupe qui matérialise des interfaces. En comparaison avec la réflexion, la programmation par aspects propose des langages plus adaptés à la description des adaptations : le programmeur d'aspect raisonne par rapport au déroulement et à la structure du programme de base, alors que la réflexion porte sur l'interprétation du programme. Cependant, les implémentations des systèmes dynamiques de programmation par aspects pour le langage C sont rares. De plus, lorsque les langages d'aspect proposés ne sont pas limités, ce sont les techniques de tissage qui requièrent l'utilisation sur le système de base d'un compilateur

modifié.

Pour ces raisons, nous proposons d'évaluer l'efficacité d'un système de programmation par aspects par réécriture de code à la volée pour l'évolution des systèmes patrimoniaux. La programmation par aspects offre un confort de conception des adaptations sans préjuger de la conception du système de base. Et, les techniques de réécriture de code permettent une adaptation fine et au plus tard du logiciel, tout en offrant des performances optimales.

Aucune règle n'existe, les exemples ne viennent qu'au secours des règles en peine d'exister.

André Breton

Chapitre 3

Problèmes illustrateurs

Où nous introduisons par des exemples concrets et variés sur des applications réelles, la problématique de l'évolution des applications systèmes patrimoniales. La description de plusieurs exemples nous permet ici d'ébaucher les besoins de l'évolution des systèmes informatiques. Ces exemples seront réutilisés tout au long du document : d'abord pour illustrer le langage d'aspect que nous avons conçu, puis pour évaluer les performances de notre prototype.

Sommaire

3.1 Correction d'erreurs – Trous de sécurité **28**
3.1.1 Exploitations de l'allocateur dynamique de mémoire 31
3.1.2 Débordements de tableaux 34
3.1.3 Interblocages de verrous 35
3.2 Ajout d'une politique de préchargement dans Squid **36**
3.3 Adaptation **37**
3.3.1 Supervision de l'exécution de Squid 37
3.3.2 Remplacement de TCP par UDP 38
3.4 Bilan **39**

La problématique traitée dans cette thèse est celle de l'évolution logicielle des applications systèmes patrimoniales. Pour ces systèmes, deux difficultés sont à lever. Premièrement, ces applications sont le fruit du travail de nombreux développeurs et ce parfois sur plusieurs années. La qualité de leur code doit donc être préservée. Deuxièmement, ces applications sont soumises à des contraintes fortes de performances.

L'état de l'art présenté dans le chapitre précédent, nous a amené au constat que les approches existantes de l'évolution logicielle étaient inadaptées aux applications systèmes patrimoniales. De cet échec, nous avons dégagé trois raisons : une adaptabilité non anticipée limitée, des performances dégradées et une complexité trop importante pour la conception des adaptations. Nous en avons découlé notre proposition : l'utilisation d'un langage de programmation par aspects pour le langage C et du tissage par réécriture à la volée du code exécuté par le processeur.

Le choix de l'instrumentation à la volée du code binaire est motivé par les performances et les langages d'aspect pour la facilité de développement des adaptations. En effet, les langages d'aspect associent du code à déclencher en réaction à des événements

de l'exécution d'un programme de base. Le choix de ces événements doit donc être réalisé intelligemment puisqu'il va déterminer l'expressivité de notre langage d'aspect.

Pour nous diriger dans ce choix, nous proposons d'analyser des exemples d'évolutions d'applications système. Cette méthodologie doit néanmoins être appliquée avec parcimonie : un choix peu varié d'exemples limiterait l'expressivité du langage, tandis qu'un nombre important d'exemples ferait tendre le langage vers un « fourre-tout » difficile d'utilisation. Lehman [BL76] définit quatre types d'évolutions logicielles. *(i)* La correction d'erreurs est à l'origine de la majorité des mises-à-jour logicielles et est une motivation d'autant plus importante qu'on constate que le nombre d'erreurs dans un logiciel est constant dans le temps [BL76]. *(ii)* L'intégration de nouvelles fonctionnalités est certainement la plus importante motivation de l'évolution logicielle tant elle conditionne l'utilité d'un système vis-à-vis des utilisateurs. *(iii)* L'adaptation à l'environnement est nécessaire, par exemple pour le support de nouveaux matériels ou lors du changement des interfaces d'une librairie tierce. *(iv)* La restructuration qui est d'autant plus importante qu'un logiciel tend invariablement à se complexifier [Pre04]. Ce dernier type est néanmoins limité à l'évolution à froid.

Nous proposons d'utiliser des exemples de chaque type d'évolutions pour motiver les primitives de notre langage d'aspect. Les exemples que nous avons choisis portent sur des applications réelles : le cache Web Squid, la librairie standard C, le noyau Linux, *etc*, et concernent des problématiques concrètes : sécurité, supervision, performances. Pour chaque exemple, nous présentons son contexte et sa problématique, et nous en déduisons les éléments nécessaires à leurs implémentations. Les exemples discutés ici seront réutilisés tout au long de ce document ; d'abord dans le chapitre 4, où nous présenterons leurs implémentations pour illustrer notre langage d'aspect ; puis dans le chapitre 6, où ces exemples serviront à l'évaluation de la faisabilité et des performances de notre approche.

Pour illustrer la correction de logiciel, notre premier exemple discute de la correction de trous de sécurité dans les applications C. Nous présentons ensuite l'intégration d'une politique de préchargement dans le cache Web Squid. Enfin, nous présentons la difficulté de la supervision d'une application système et du remplacement d'un protocole de communication réseau.

3.1 Correction d'erreurs – Trous de sécurité

L'histoire de l'internet débute en 1969 par ARPANET, un projet financé par l'*Advanced Research Projects Agency*. Ce projet avait pour but de créer un réseau de communication fonctionnel en cas de destruction partielle. Ce n'est qu'en 1988 que la communauté informatique prend conscience de la sévérité de la menace informatique après que le vers *Morris* ait causé plus de dix millions de dollars de pertes d'après l'*US Government Accountability Office*. À cette époque, la menace informatique reste marginale, mais va croître exponentiellement avec la démocratisation de l'informatique et de l'internet comme le montre les données récoltées par le CERT/CC (*Computer Emergency Response Team, Coordination Center* [Cen03]) (figure 3.1). Bien que la sécurité ait pris une place plus importante dans le monde informatique, on constate toujours un accroissement exponentiel des vulnérabilités découvertes dans les systèmes informatiques comme l'atteste la figure 3.2.

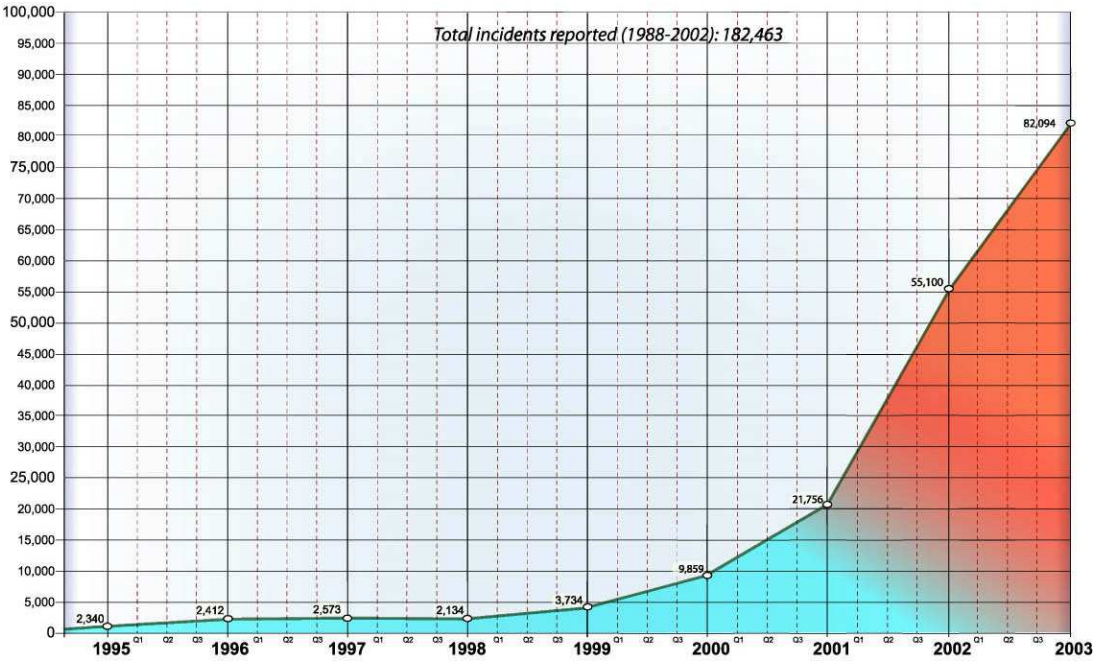


FIG. 3.1: Rapports d'incidents constatés entre 1988 et 2002 (recherches de vulnérabilité, compromissions d'identité, surveillances du trafic réseau, dénis de service, exploitations de relation de confiance, codes malicieux et attaques d'infrastructures réseaux)

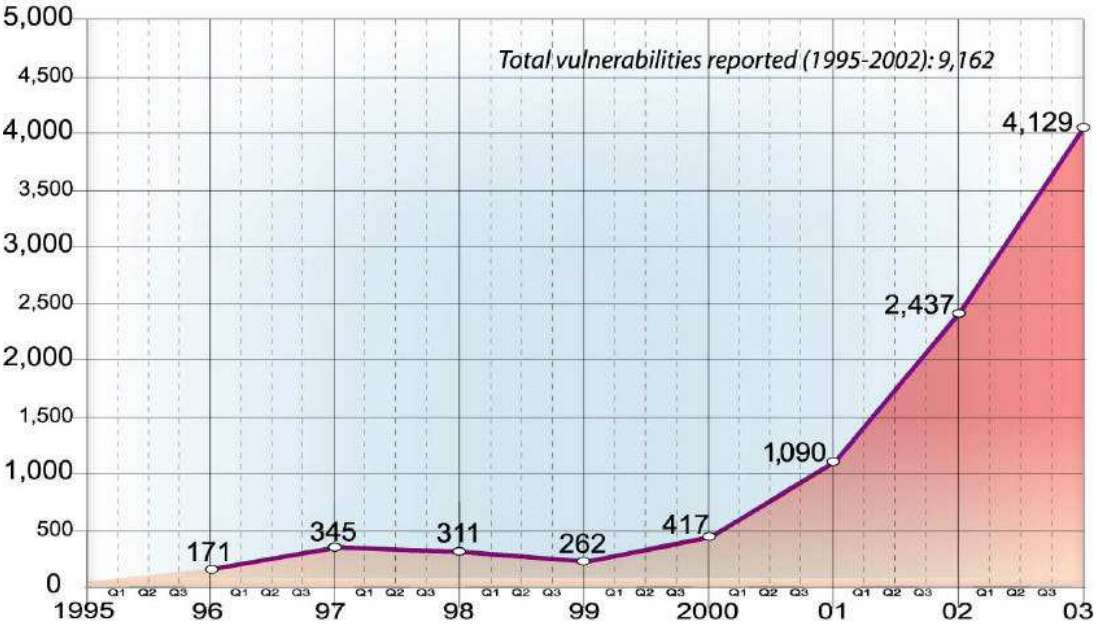


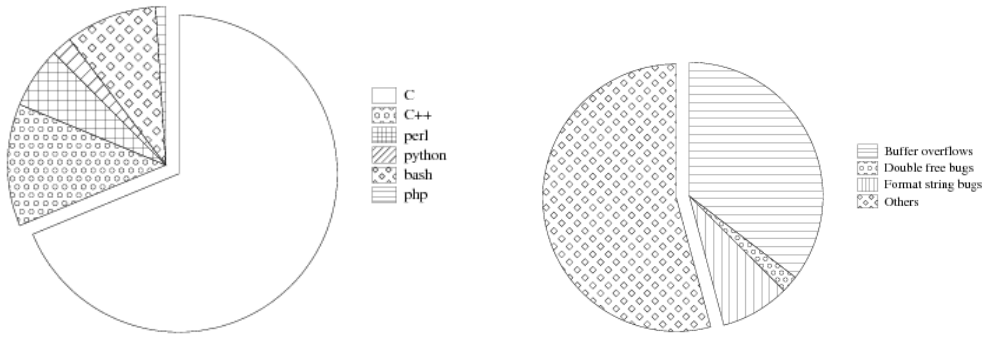
FIG. 3.2: Vulnérabilités constatées entre 1995 et 2002 (défauts de design dans les logiciels et les protocoles, défauts d'implémentations des logiciels et des protocoles, vulnérabilités dans la configurations des systèmes et des réseaux)

Depuis *Elk Cloner* (1982) considéré comme le premier virus informatique ayant eu une propagation significative [Spa94], on a pu observer les nombreuses techniques d'attaques utilisées par les pirates informatiques qu'elles soient purement techniques ou du domaine du *social engineering*. Depuis 2001, des virus tels que *Code Red* [CAI07] et *Witty* [SM04] ont mis en lumière une nouvelle génération d'attaques informatiques : les *zero-day worms*. Le point commun de ces virus est que chacun exploite une vulnérabilité dont l'existence venait d'être publiée. La tactique suivie par leurs concepteurs était d'exploiter la période entre la découverte de la vulnérabilité et la correction effective de celle-ci sur les applications en production. Leurs développeurs ont misé non pas sur la furtivité mais sur la vitesse de propagation. Appliquant cette stratégie à la lettre, le virus Witty a battu le record du plus court délai entre la publication d'une vulnérabilité et le début de la propagation du virus (l'appellation *zero-day* est en fait ici abusive pour Witty qui fût lancé le jour suivant la publication de la faille).

Ces deux tendances (l'augmentation du nombre de vulnérabilités et les attaques en *zero-day*) exacerbent la problématique de la sécurité informatique. Face à un grand nombre de vulnérabilités, il est difficile pour les développeurs de corriger leurs logiciels rapidement ; ainsi, lorsqu'un bulletin de sécurité informe de la découverte d'une vulnérabilité, un correctif n'est pas nécessairement disponible. De plus, certaines entreprises ont choisies de ne diffuser la découverte d'une vulnérabilité que lorsque celle-ci est corrigée, cette situation est bien plus alarmante puisque rien ne garantit qu'un pirate informatique n'ait pas connaissance de cette information, lui laissant d'autant plus de temps pour mettre en place une attaque.

La rapidité des attaques *zero-day* et le nombre de vecteurs d'attaques font qu'il est quasiment impossible pour un administrateur système de garantir la sécurité d'un parc informatique. La seule prévention effective est de tenir les parcs informatiques constamment à jour. Or, les méthodes et les outils de mise à jour de logiciels sont totalement inadaptés face à des attaques de type *zero-day*. En effet, les correctifs de sécurité sont pour la plupart diffusés sous forme de *patch* qui nécessitent l'arrêt et le redémarrage de l'application vulnérable. Il est inconcevable dans le cas d'un serveur de e-commerce ou d'une banque de stopper le service fourni aux clients. Par exemple, le serveur en charge des transactions VISA pour le continent nord-américain est composé de vingt et un super-calculateurs. Il prend en charge plus de trois mille transactions par seconde et doit atteindre un taux de disponibilité de 99,5% sur une année [Pes00]. Nous pensons qu'un tisseur d'aspect à la volée garantissant à la fois la continuité de service et des performances optimales est une contribution au domaine de la sécurité informatique.

Nous avons analysé une partie des bulletins d'alerte du site de la distribution Linux Debian [Deb]. *Debian Security* a diffusé plus de 1300 bulletins d'informations depuis 1997. Sur ces 1300 bulletins, nous avons examiné un échantillon de plus d'une centaine de bulletins (cet échantillon correspond aux bulletins diffusés sur le premier trimestre de 2005). Nous avons constaté comme le montre la figure 3.3(a) que 70% des vulnérabilités découvertes dans les logiciels libres, concernent des programmes écrits en C et 12% des programmes en C++. Nous nous sommes intéressés aux vulnérabilités du langage C. Comme le montre la figure 3.3(b), 8% sont des exploitations de chaînes de format des fonctions à paramètres variables, 2% concernent des exploitations de l'allocateur de mémoire de la librairie standard C et environ un tiers (35%) sont des débordements de tableaux. Nous avons choisi d'étudier plus en détail ces deux dernières failles (allocateur



(a) Répartition par langages de programmation

(b) Répartition par types de vulnérabilités dans les logiciels écrits en C

FIG. 3.3: Vulnérabilités dans les logiciels libres. Source : *Debian Security* pour le premier trimestre 2005

mémoire et débordement de tableau).

3.1.1 Exploitations de l'allocateur dynamique de mémoire

Les systèmes Unix organisent la mémoire des processus utilisateurs en trois parties. La zone « .text » qui contient le code exécutable, la pile qui maintient les informations nécessaire à l'exécution du code, et le tas qui permet aux utilisateurs de stocker les variables du programme. La quantité de données traitée par un programme étant variable au cours de l'exécution d'un programme, les systèmes Unix implémentent un appel système nommé *brk* qui permet de redimensionner la taille du tas. Cet appel système ne fournit aucun moyen de suivre l'utilisation mémoire du processus et ne permet pas de connaître les zones mémoires utilisées et celles libres. La norme POSIX a défini une interface standard pour l'allocation dynamique de mémoire. L'implémentation de ce standard pour le C fournie par la librairie standard C, propose un allocateur de mémoire dynamique. Cet allocateur de mémoire utilise l'appel système *brk* pour allouer de grands fragments de mémoire qu'il partage ensuite en plus petits fragments. Le développeur dispose d'une interface (fonctions *malloc*, *free*, *etc*) lui permettant ainsi de manipuler facilement et à grain fin le contenu du tas.

Les performances d'un allocateur dynamique de mémoire s'évaluent sur trois critères : la complexité des algorithmes d'allocations et de libérations de la mémoire, le rapport de grandeur entre la mémoire totale disponible et la mémoire utile, et le taux de fragmentation de la mémoire. Ces critères ont déterminé les choix architecturaux de l'allocateur de mémoire de la librairie standard C au détriment de la sécurité.

La librairie standard C organise la mémoire en deux listes doublement chaînées : la première liste tous les blocs mémoires et la seconde uniquement les blocs libres. Afin de maximiser la mémoire utile, les données nécessaires au fonctionnement de l'allocateur (listes des fragments utilisés/inutilisés) sont stockées sur le tas avec les données du programme. Le listing 3.1 montre l'organisation des fragments mémoires. Chaque fragment contient sa taille et celle du fragment précédent (ce sont des tailles totales et

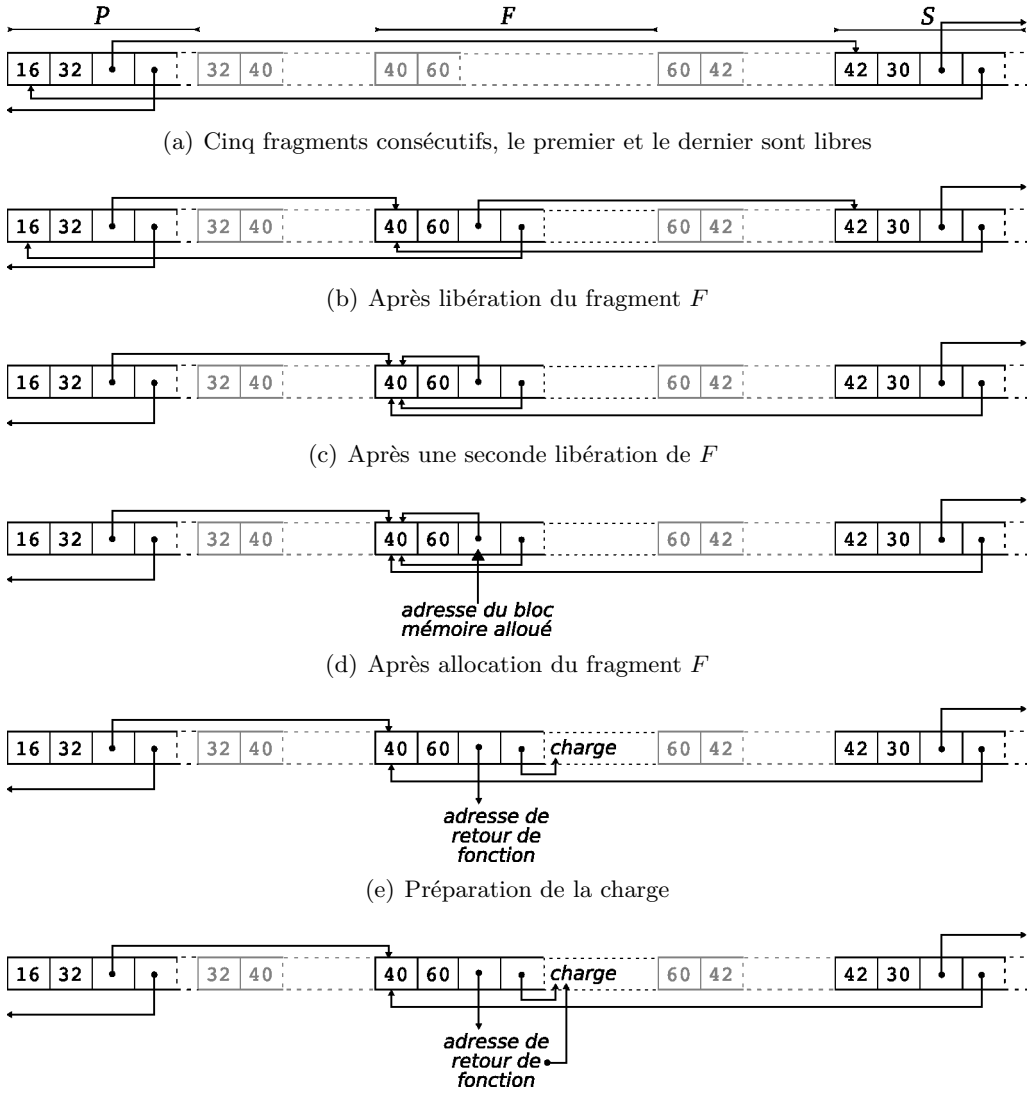
```
typedef struct chunk {
    uint32_t prev_size; // size of previous adjacent chunk
    uint32_t size;  // size of the chunk
    uint32_t* next; // next free chunk
    uint32_t* prev; // previous free chunk
} chunk_t;

void* allocate (chunk_t* c) {
    c->prev->next = c->next;
    c->next->prev = c->prev;

    return &(c->next);
}

/* p is the free chunks after which to insert c */
void free (chunk_t* c, chunk_t* p) {
    c->next = p->next;
    c->prev = p;
    p->next->prev = c;
    p->next = c;
}
```

Listing 3.1: Définition des listes doublement chaînées utilisées par la librairie stantard C pour l'allocation dynamique de mémoire

non utiles), comme les fragments sont consécutifs en mémoire, ces deux informations permettent de parcourir la liste des fragments dans les deux sens à partir de n'importe quel fragment. Lorsque le fragment est alloué, le reste du fragment contient la mémoire utile pour l'utilisateur, sinon, le fragment contient également les adresses du fragment libre suivant et du fragment libre précédent. Le listing 3.1 présente également les routines réalisant le maintien des listes chaînées lors de l'allocation et la libération de fragments.

La librairie standard C ne vérifie pas lors de la libération d'un fragment que celui-ci était alloué. Ce comportement peut être exploité à des fins malicieuses. La figure 3.4(a) montre une portion de la mémoire, on y voit cinq fragments, deux sont libres : P et S. Lorsque que le fragment F est libéré, celui-ci est chaîné entre les fragments P et S (figure 3.4(b)). En forçant le programme à libérer le fragment F une seconde fois (figure 3.4(c)), l'allocateur va à nouveau l'insérer dans la liste entre P et S. Or comme F se trouve déjà entre P et S, les éléments *next* et *prev* de F vont pointer sur lui-même (listing 3.1 lignes 17 et 19). Comme F suit toujours P, il est toujours possible d'allouer F. Mais comme les éléments *next* et *prev* de F pointent sur lui-même, l'allocation de F laisse la liste des fragments libres inchangées et retourne l'adresse de l'élément *next* de F (figure 3.4(d)). Si le fragment F est destiné à recevoir une entrée clavier ou réseau, un utilisateur malveillant peut facilement écraser *next* avec l'adresse d'une charge virale stockée dans la suite de l'entrée, et *prev* par une adresse sur la pile pointant l'adresse de retour d'une fonction (figure 3.4(e)). En forçant le programme à allouer de nouveau le fragment F (figure 3.4(f)), la librairie standard va écraser l'adresse de retour de la fonction avec l'adresse de la charge virale (listing 3.1 ligne 10). Il suffira alors à l'utilisateur malveillant d'attendre que cette fonction termine pour que la charge virale s'exécute.

Ce comportement est une faille de sécurité importante qui a été largement exploitée par le passé pour conduire des attaques de déni de service [AL03] ou pour prendre le contrôle de systèmes à distance [Des97]. Afin de se prémunir de ce type d'attaques, une implémentation sécurisée de la librairie standard C peut être choisie lors du chargement des processus. Néanmoins, l'expérience démontre que cette variante de l'allocateur dynamique de mémoire est nettement moins performante et rarement utilisée. Ainsi, un



(a) Cinq fragments consécutifs, le premier et le dernier sont libres

(b) Après libération du fragment F

(c) Après une seconde libération de F

(d) Après allocation du fragment F

(e) Préparation de la charge

(f) Une nouvelle allocation de F finit la mise en place de l'attaque

FIG. 3.4: Réalisation d'une attaque de type « double free bug »

administrateur système faisant face à une telle faille de sécurité, devrait alors déployer une version corrigée de l'application fautive. Ceci nécessitant de stopper l'application. Bien que le cache Web Squid utilise son propre allocateur de mémoire pour des raisons de performances, celui-ci se trouve également vulnérable aux attaques par exploitation de « double free bug » [Ubu05]. La gestion dynamique de la mémoire est un aspect crosscuttant dans Squid, en effet, elle représente environ 71% du code de Squid. Ainsi, pour garantir la continuité de service d'une application dans le cas d'une faille de type « double free bug », tout en limitant l'impact sur les performances, nous pensons que la protection contre l'exploitation de cette vulnérabilité doit pouvoir être activée uniquement en cas de nécessité.

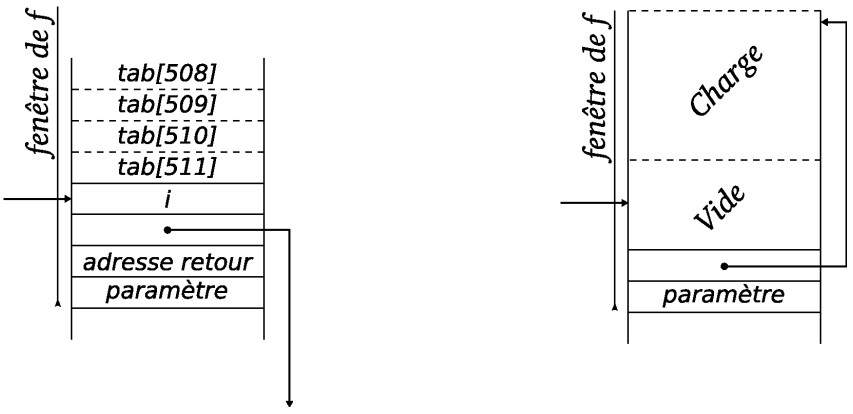
> La correction d'une faille de sécurité comme le « double free bug » nécessite de modifier la manière dont un programme interagit avec une bibliothèque. Ici, ces interactions sont principalement des appels aux fonctions d'allocation (*malloc*, *calloc*, *etc*) et de libération (*free*) mémoire.

3.1.2 Débordements de tableaux

Dans le langage C, il n'est possible de connaître la taille d'un tableau que lors de son allocation. D'après les normes ISO et ANSI [Ins99], un accès en dehors des limites d'un tableau ne provoque pas une erreur immédiate, mais un comportement dépendant de l'implémentation. Les débordements de tableaux entraînent généralement la corruption des données adjacentes en mémoire. Comme l'allocation des tableaux est faite soit sur le tas soit sur la pile du processus, un débordement peut entraîner la corruption des données du programme ou des informations de contrôle de l'exécution. Comme le montre la figure 3.5, lorsque le tableau est destiné à stocker une entrée clavier ou réseau et que le tableau se trouve sur la pile (variable locale), il est possible de modifier l'adresse de retour d'une fonction de telle sorte qu'elle pointe vers une charge stockée dans le tableau. Lorsque la fonction dont l'adresse de retour a été corrompue se termine, le programme va déclencher la charge.

Ce comportement a été largement exploité par les pirates informatiques pour contourner la sécurité des systèmes [WK03]. Il est donc crucial que les développeurs C vérifient que tout accès à un tableau soit valide. Néanmoins, cela implique d'ajouter du code comparant l'index de l'élément accédé avec la taille du tableau. Or, ce type de code n'est pas forcément trivial à implémenter : les pointeurs complexifient cette tache. De ce fait, des travaux proposent d'implémenter ces vérifications au niveau du compilateur [JK97, RL04]. Malheureusement, ce type d'approches implique un surcoût important à l'exécution : la plus performante de ces implémentations (CRED [RL04]) génère un ralentissement de 30% sur l'exécution de programmes usuels. De plus, les compilateurs comme GCC [Foua] et Intel C Compiler [Cor] ne proposent que depuis récemment voire pas du tout cette fonctionnalité.

Ainsi, un administrateur système devant traiter un débordement de tableau n'a d'autre choix pour garantir la sécurité que de stopper toutes les instances de cette application et de la mettre à jour afin d'utiliser une version plus sûre. Dans la pratique, la plupart des administrateurs choisissent sciemment d'ignorer ces failles de sécurité. En effet, le redémarage d'une application induit trois problèmes. Premièrement, cela interrompt la continuité de service. Deuxièmement, le travail en cours est perdu lors



(a) Fenêtre d'une fonction f, f possède deux variables locales : i et le tableau *tab* de 512 éléments

(b) Injection de la charge et écrasement de l'adresse de retour

FIG. 3.5: Exploitation d'un débordement de tableau

de l'arrêt. Troisièmement, cette solution est inadéquate dans le sens où il n'est pas nécessaire de protéger tous les accès à tous les tableaux. Il faut garantir une sécurité suffisante pour décourager les pirates informatiques. Ainsi, les vérifications de bornes sur les accès à des tableaux ne devrait être utilisées que lorsque l'environnement est hostile.

La vérification de borne dans un programme C est une fonctionnalité crosscutante. En effet, une fonction C prenant en argument un tableau, quand elle est correctement écrite, prend un second argument : la taille du tableau. Du code de vérification de borne existe dans les 104 fichiers C qui composent Squid. Sur les 57635 lignes de code de Squid, au moins 485 concernent de la vérification de bornes de tableaux.

Pour corriger ce type d'erreur, il est donc nécessaire de modifier la manière dont un programme manipule les tableaux, or les tableaux se manipulent soit via un index par rapport à leur base, soit par des pointeurs.

3.1.3 Interblocages de verrous

Les « deadlocks » sont un problème bien connu des développeurs de programme concurrents. Un interblocage de verrous survient lorsqu'au moins deux files d'exécution attendent mutuellement que l'autre libère une ressource pour continuer leurs exécutions. Pour des raisons d'efficacité, la plupart des applications réelles sont développées en C. Le langage C ne proposant pas de facilités pour la programmation concurrente, les primitives basiques de synchronisation (processus légers, verrous, *etc*) sont fournies par des librairies. Cette approche ne fournit que peu voire pas de garanties sur la correction d'un programme. Par exemple, les opérations sur les verrous fournies par la librairie des processus légers POSIX (*libpthread*) ne propose que deux vérifications dynamiques sur l'usage des verrous. *(i)* aucun processus léger ne pourra acquérir un verrou si l'accès à celui-ci est déjà accordé, *(ii)* seul le processus léger ayant l'accès à un verrou pourra le libérer. Ces deux propriétés sont insuffisantes pour garantir qu'un

programme ne contienne pas d'interblocage. Bien que les analyseurs statiques de code permettent de détecter ce type de problèmes, ceux-ci sont incapables de traiter des applications de grande taille. Ceci pose un problème important pour la réutilisation des librairies de programmation concurrentes.

Dans le but de résoudre le problème des « deadlocks », plusieurs solutions à base d'analyses dynamiques ont été proposées. Une approche basée sur les temps d'inactivité : lorsqu'un processus léger est inactif sur une durée arbitraire, il est considéré bloqué. Cette approche a l'avantage de n'impliquer qu'un faible surcoût à l'exécution d'un programme, néanmoins, n'étant pas complète, elle peut entraîner le rapport de faux positifs. De plus, lors de la détection d'un interblocage, cette solution ne fournit aucune information permettant de résoudre le blocage : quel est l'ensemble minimal de processus léger à terminer ? Une autre approche consiste à maintenir une représentation des réservations faites par les processus légers dans un graphe des ressources allouées. Les processus légers et les ressources y sont représentés par des nœuds ; Les arcs orientés d'un processus léger vers une ressource indiquent que le processus léger est en attente d'obtenir l'accès à cette ressource ; Et les arcs orientés d'une ressource à un processus léger indiquent que la ressource est affectée au processus. Dans un graphe des ressources allouées, un cycle indique un interblocage. Cette approche présente l'avantage d'être complète, néanmoins, elle nécessite de suivre l'utilisation des primitives de synchronisation. De plus, l'algorithme de détection de cycle dans un graphe de ressources allouées a une complexité en $O(n+m)$ (n étant le nombre de processus légers, et m le nombre de ressources).

> Cet exemple souligne également la nécessité d'adapter les interactions entre un programme et une librairie. Néanmoins, ces interactions ne peuvent être prises en compte individuellement mais dans le cadre d'un protocole : la libération d'un verrou n'est valide que si elle est réalisée par son possesseur (processus, thread, *etc*).

3.2 Ajout d'une politique de préchargement dans Squid

Les caches Web sont des intermédiaires réseau qui enregistrent le contenu Web téléchargé par les utilisateurs d'un réseau. Lorsqu'un utilisateur requiert un contenu ayant été mémorisé, le cache Web le délivre à la place du serveur distant. Les buts des intermédiaires sont multiples. D'une part, le cache Web améliore la tolérance aux pannes : en cas de coupures réseaux, une partie des contenus est toujours disponible. D'autre part, la bande passante consommée diminue : un contenu mémorisé n'a pas besoin d'être récupéré. Finalement, les caches Web diminuent les délais (latence) perçus par les utilisateurs pour accéder aux contenus distants.

En ayant connaissance de la sémantique du contenu Web téléchargé, un cache Web peut encore améliorer la latence perçue par les utilisateurs en préchargeant du contenu en fonction des requêtes précédentes. Par exemple, lors du téléchargement d'une page Web, il peut précharger les pages et les images référencées par celle-ci. Ces techniques de pré-chargement se distinguent par la manière dont elles prédisent les requêtes futures. Les « oracles » se servant d'informations réparties en plusieurs points, leur implémentation empêche l'encapsulation de la fonctionnalité de pré-chargement. De plus, il est très improbable qu'il existe un oracle universel [IBCM00]. Un module de pré-chargement lié statiquement est donc inadéquat.

Coady *et al.* ont mis en évidence ce problème de modularisation du pré-chargement dans FreeBSD [CKFS01]. Ségura-Devillechaise *et al.* ont également montré que ce problème pouvait s'exprimer dans un langage d'aspect sous la forme d'une construction de type flot de contrôle [SDMML03].

Bien que le préchargement ait pour objectif de diminuer le temps de latence lors de la consultation de contenu Web, il augmente l'utilisation des ressources locales : la bande passante et le stockage sur disque. Lorsque la demande de ressource est trop importante, le préchargement rentre en concurrence avec les requêtes des utilisateurs. Dans de telles situations, le préchargement doit être désactivé temporairement. Le cache Web Squid utilise les descripteurs de fichiers Unix pour gérer les connections réseaux. Or, le nombre de descripteurs de fichiers utilisés est limité par le système d'exploitation. En comparant le nombre de descripteurs utilisés par Squid avec la limite définie par le système, il est possible d'estimer quand le préchargement doit être désactivé.

De part l'architecture événementielle de Squid, l'ajout d'une politique de préchargement nécessite de modifier plusieurs événements liés séquentiellement.

3.3 Adaptation

L'adaptation dynamique d'un logiciel est une phase importante de son cycle de vie, elle requiert de superviser son exécution pour effectuer les adaptations nécessaires à son bon fonctionnement. Ici, nous présentons deux exemples, le premier illustre la problématique de la supervision d'une application système telle que Squid, le second montre comment adapter un protocole de communication réseaux utilisé par Squid.

3.3.1 Supervision de l'exécution de Squid

La supervision de l'exécution d'un logiciel est un point important pour l'adaptabilité d'un logiciel. En effet, la supervision est nécessaire lors du développement pour le débogage, et en production pour mieux l'adapter à son environnement. Par exemple, la politique de préchargement décrite précédemment, est conditionnée par l'état du système : la politique n'est activée que lorsque le système n'est pas surchargé.

Superviser l'exécution d'un logiciel tel que Squid n'est pas chose aisée. D'une part, l'architecture événementielle de Squid rend son code source peu lisible et son exécution non linéaire. D'autre part, Squid gère lui même la plupart des ressources (réseaux, disques, mémoire) du système sans passer par des librairies standard. Ainsi, la supervision ne peut se faire uniquement en observant de l'extérieur l'exécution de Squid.

Il est impossible de superviser en permanence toutes les données et événements générés par un tel logiciel, le code de supervision serait alors tellement imposant que l'exécution ne serait plus représentative. Comme l'exécution d'un cache Web n'est pas reproductible (le comportement du cache dépend fortement du réseau) et que chaque expérience sur Squid demande un travail fastidieux, il faut pouvoir ajouter et retirer dynamiquement le code de supervision.

Bien que les logiciels système comme Squid gèrent eux-même la plupart des ressources en contournant le système d'exploitation et les librairies standard, la seule observation d'un logiciel système n'est parfois pas suffisante pour capturer son état.

Monitorer l'utilisation faite par Squid des disques durs pour chaque client, illustre cette problématique. Squid utilise à la fois la mémoire vive et l'espace disque comme cache. La mise en cache est transparente pour Squid, elle est faite dans un *mapping* anonyme qui en fonction des pages mémoire actuellement chargées dans la TLB résulte soit à une écriture en mémoire soit à une écriture sur disque. Néanmoins l'écriture en elle-même est faite par le système. Lors de l'écriture, le système ne sait pas de quel client de Squid provient la requête associée. Aussi, pour comptabiliser les écritures par clients dans Squid, il est impératif d'observer les exécutions simultanées de Squid et du système.

Cet exemple montre qu'un code d'adaptation n'est pas limité à une application. Il est donc nécessaire d'envisager un langage d'adaptation raisonnant sur l'exécution simultanée de plusieurs applications.

3.3.2 Remplacement de TCP par UDP

HTTP est un protocole applicatif bâtit au dessus de TCP, un protocole de transport en mode connecté. Bien que la majorité des pages Web n'excèdent en taille les huit kilo-octets [AJ00], le total transféré lors de la consultation d'une page Web est largement dominé par les données de contrôle du protocole TCP. Ceci montre que le protocole TCP est inapproprié aux petits échanges et aux connections de courtes durées. Bien que la version 1.1 de HTTP introduise les connections persistantes pour permettre aux clients de consulter plusieurs pages d'un même serveur à travers une unique connection TCP, le nombre de connections TCP simultanées est limité au niveau du système d'exploitation, poussant les serveurs de contenu Web à fermer les connections au plus tôt. Ainsi, plusieurs études promulguent le remplacement de TCP par UDP pour le transport de petits contenus Web [CM03, CGRS90, RW01]. Malgré l'amélioration immédiate des performances des échanges qu'impliquerait une telle solution, celle-ci requiert de redémarrer les serveurs de contenu pour utiliser le nouveau protocole. Arrêter un serveur de e-commerce peut coûter un montant important. Ainsi, pour que la solution de remplacement de TCP par UDP soit viable, celle-ci devrait pouvoir être déployée sur demande et à la volée.

Le remplacement de TCP par UDP dans une application est difficile. En effet, le choix d'un protocole de transport impose des contraintes sur le développement de l'application qui sont difficiles à remettre en question ultérieurement. Il est également impossible de réduire ces contraintes en un seul bloc de code. Par exemple, malgré les efforts de modularisation, l'interface TCP est utilisée dans 7 des 104 fichiers source de Squid.

Les librairies de communication telles que nous les connaissons aujourd'hui, furent introduites dans le système BSD 4.2 en 1983. C'est cette version qui successivement remaniée par de nombreux contributeurs dont le « Computer Systems Research Group » de l'Université de Berkeley et le laboratoire AT&T est aujourd'hui largement répandue dans les systèmes Unix. Comme le montre la figure 3.6, les librairies de communication TCP et UDP sont articulées autour de fonctions C qui doivent être appelées séquentiellement. Dans un programme C, une connection TCP doit en premier lieu être établie (*socket*, *connect*, *bind* and *listen*). Les fonctions *read* et *write* permettent ensuite d'échanger des données sur la connection qui sera terminée par un appel à

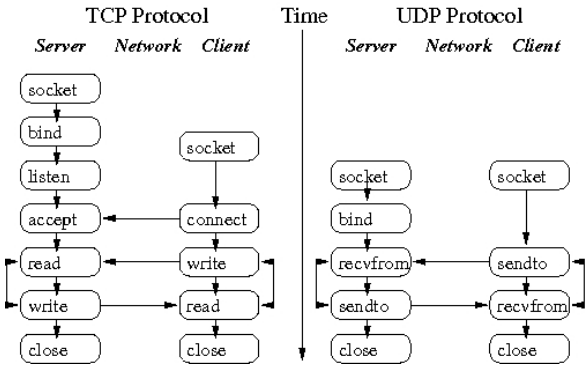


FIG. 3.6: Utilisation des interfaces TCP et UDP

close. L'interface UDP est très similaire. L'initialisation d'un socket se fait par les fonctions *socked* et *bind*, les échanges utilisent les fonctions **recvfrom** et **sendto** et la terminaison de la connection est identique à TCP (fonction **close**). On note que seules certaines phases de l'utilisation des protocoles de transport TCP et UDP utilisent les mêmes fonctions (*socket*, *bind* et *close*). Ceci est dû à la volonté de séparer l'utilisation des protocoles de transport du protocole réseau utilisé (par exemple IP). Changer le protocole utilisé dans un programme C n'est pas simple, en effet, les fonctions décrites précédemment sont utilisées sur l'ensemble du code source de l'application. De plus, l'ordre d'invocation de ces fonctions est important pour le passage en UDP.

> Cet exemple illustre parfaitement les interfaces basées sur des protocoles d'utilisation. Quand de telles interfaces sont mal utilisées, il devient impossible de faire évoluer le programme. Il est donc nécessaire de pouvoir raisonner sur ces interfaces pour exprimer des évolutions logicielles.

3.4 Bilan

Dans ce chapitre, nous avons motivé la problématique de l'adaptation des applications systèmes pratrimoniales au travers de plusieurs exemples d'évolutions logicielles. Nous avons choisi des exemples d'évolutions couvrants l'ensemble des besoins d'évolutions dynamiques : l'intégration de nouvelles fonctionnalités, la correction d'erreurs, la supervision et le débogage, et l'adaptation. Pour chaque exemple, nous avons présenté le contexte de ces évolutions et nous avons motivé la nécessité de leurs intégrations pendant l'exécution. Ces exemples s'appuient sur des problématiques concrètes et s'appliquent à des systèmes informatiques largement diffusés : Squid, Linux, wu-ftpd, la librairie standard C, *etc.*

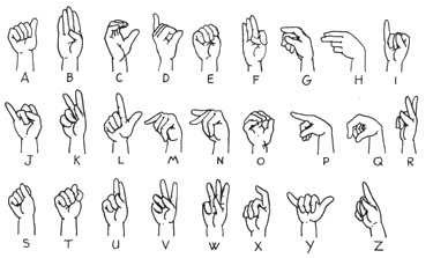
Nous avons pu au travers de ces exemples dégager les éléments nécessaires à l'évolution dynamique des systèmes patrimoniaux. Nous avons notamment constaté que les besoins étaient comportementaux et non pas structurels. Par exemple, les évolutions examinées ne requièrent pas de modifier des structures de données des programmes. Les exemples ont principalement mis en lumière la nécessité d'adapter les interactions entre les programmes et les bibliothèques. Ces interactions se matérialisent par des

appels de fonctions et des échanges de données qui sont liés par des protocoles d'utilisation. Ces protocoles se matérialisent par l'exécution séquentielle ou imbriquée de plusieurs appels de fonctions et/ou échanges de données. Les exemples visités ont également souligné la nécessité de prendre en compte les interactions entre plusieurs flots d'exécutions : processus léger, processus, applications/noyau.

Dans la suite de ce document, nous verrons comment ces observations ont influencées la conception de notre langage d'aspect et l'implémentation de notre système de tissage.

Deuxième partie

Contribution



Chapitre 4

Un langage pour l'adaptation

Où nous proposons un langage d'aspect proche des concepts du langage C. Ce langage raisonne sur les événements d'exécution des programmes C : appels de fonctions, accès à des variables globales ; et sur l'enchaînement séquentiel ou imbriqué de ces événements [DFL+06, DFL+05, LSDFM06]. Il permet également l'adaptation des interactions entre plusieurs applications [LM07]. Nous y montrons également comment ce langage permet l'expression concise des problématiques d'évolutions (ajout de fonctionnalités, correction de bogues, etc) dans des applications réelles (Squid, Linux, etc) présentées dans le chapitre précédent.

Sommaire

4.1 Choix des points de jonction **44**
4.1.1 La relation code C – code machine 45
4.1.2 Les points de jonction 47
4.2 Le langage de coupe **48**
4.2.1 Sélecteurs de points de jonction 49
4.2.2 Variables de coupe 49
4.3 Le langage d'aspect **50**
4.3.1 Aspects 50
4.3.2 Placement des aspects 51
4.4 Exemples illustrateurs **52**
4.4.1 Concepts de base – Exploitation de l'allocateur de mémoire C 53
4.4.2 Les flots de contrôle – Préchargement dans le cache Web Squid 53
4.4.3 Les séquences – Remplacement de TCP par UDP 54
4.4.4 Les accès aux variables – Les débordements de tableaux . . 55
4.4.5 *bind* – Les interblocages de verrous 56
4.4.6 Placement des aspects – La supervision du cache Web Squid 57
4.5 Conclusion **57**

Les méthodologies de conception et de programmation poursuivent principalement un même objectif : partitionner les logiciels en modules indépendants gérant chacun une fonctionnalité. On trouvera alors des modules métiers ainsi que des modules techniques comme la journalisation ou l'authentification des utilisateurs. Dans la pratique, il est difficile voire impossible d'obtenir un partitionnement complet en suivant une découpe en modules. En effet, les différentes fonctionnalités d'un logiciel sont

amenées à s'entrecroiser : le code métier utilise les modules techniques (*crosscutting*). Ainsi, un module dédié à une tâche se trouve fortement lié à des modules techniques ; La figure 2.5 illustre la contamination du module Tomcat d'Apache par le module de trace.

La programmation par aspects, *Aspect Oriented Programming* [KLM+97] en anglais, est un paradigme de programmation permettant de séparer les considérations métiers des considérations techniques par la modularisation des préoccupations transversales. La programmation par aspects s'inscrit en complément des paradigmes classiques et n'est pas astreinte à un langage de programmation qu'il soit impératif, objet ou fonctionnel. En programmation par aspects, une fonctionnalité transversale est spécifiée de manière autonome dans un aspect. Les aspects définissent également l'ensemble des points où ils interagissent avec le reste du code (points de jonction). Ces points de jonction ne sont pas énumérés (ce qui reviendrait uniquement à déplacer le problème de l'entrecroisement) mais définis grâce à des expressions rationnelles (le langage de coupe) précisant les points de jonction de l'aspect avec le reste du système. Comme tout autre système de modularisation (compilation pour les unités de compilation, éditions des liens pour les librairies, *etc*), les aspects sont liés au reste du système par un tisseur d'aspect.

La conception d'un langage d'aspect est donc principalement fondée sur le choix des événements du programme de base pouvant être interceptés par le langage de coupe. Le choix de ces événements détermine l'expressivité du langage d'aspect, il est limité par la faisabilité de la mise en œuvre. Lors de la définition du langage d'Arachne, nous avons mis l'accent sur deux objectifs. Le premier était de produire un langage rapide à prendre en main, nous avons donc choisi de rester proche de la syntaxe du C. Le second objectif était de proposer une implémentation performante, ainsi nous avons pris soin de restreindre les constructions du langage, notamment les séquences d'aspect, afin d'obtenir un compromis adéquat entre expressivité et efficacité.

Dans cette partie, nous présentons le langage d'Arachne qui instancie le modèle formel EAOP [DFS02]. Dans un premier temps, nous introduirons le modèle de points de jonction, c'est-à-dire les événements du programme pouvant être utilisés par le langage de coupe. Ensuite, nous présenterons le langage de coupe et la sémantique du langage d'aspect. Finalement, nous clôturerons la présentation du langage en l'illustrant au travers des exemples présentés dans le chapitre précédent.

4.1 Choix des points de jonction

Dans cette thèse, nous nous proposons de concevoir un système de programmation par aspects par réécriture de code. Nous avons dit que l'expressivité d'un langage d'aspect est directement liée au choix des points de jonction. Comme notre tisseur travaille sur du code exécutable, nous pourrions choisir notre modèle de points de jonction de sorte qu'il couvre l'ensemble des instructions processeurs. Le code assembleur étant peu structuré et obscur pour la plupart des utilisateurs de langages de haut niveau, l'utilisation d'un langage d'aspect raisonnant au niveau assembleur serait difficile d'utilisation.

Ainsi, et bien que le paradigme de programmation par aspects soit indépendant des langages de programmation, nous avons choisi de concevoir un langage d'aspect

qui soit le plus proche du langage cible, le C. Le C est un langage impératif et procédural ; les programmeurs C manipulent des données séquentiellement et partitionnent les traitement par procédures. Les séquences d'exécution des traitements des données forment un historique de l'exécution. Aussi pour concevoir notre langage d'aspect le plus proche possible du C, nous avons voulu orienter notre langage vers ces 3 axes : la manipulation des données, des procédures, et de l'historique de l'exécution.

Néanmoins, le choix des points de jonction de notre langage d'aspect n'est pas uniquement dirigé par le langage cible. Il est également conditionné du fait que notre technologie de tissage fonctionne à la volée. Par exemple, l'initialisation d'une variable globale C ne présente plus d'intérêt après le démarrage. De même, la conception de notre langage est soumisse à la faisabilité de la mise en œuvre, ici nous avons choisi de procéder par réécriture du code C compilé. Or, la compilation d'un programme C est un procédé destructif. Par exemple, une fonction qui serait inutilisée n'existera pas dans le code compilé. De même, l'ordre d'exécution des instructions dans les sources n'est pas nécessairement respecté par le code compilé si le code demeure équivalent.

Avant d'introduire le choix des points de jonction du langage d'Arachne, nous discuterons de la relation entre le code source C et le code machine, principal déterminant des points de jonction utilisables.

4.1.1 La relation code C – code machine

Introduit en 1972 par Dennis Ritchie, le C est un langage de programmation généraliste, impératif et procédural. Il fût développé dans le cadre restreint du système d'exploitation Unix et tend à rester le plus proche de la machine, notamment en offrant la possibilité d'accéder directement à la mémoire. Le C n'en demeure pas moins, un langage portable au niveau source, pouvant être compilé facilement (Tiny C Compiler, le plus petit compilateur C pèse moins de 100 kilo-octets [Bel04]) et qui ne requiert qu'un environnement d'exécution minimal. Ces raisons ont largement contribué au succès du C dans le développement de systèmes d'exploitation et d'applications système. Aujourd'hui encore, le C est le second langage le plus utilisé après Java et le premier par la communauté du libre [Sof07]).

Depuis 1972, de nombreuses variantes du langage C ont été introduites, parmi lesquelles, le C ANSI ou encore le C99 [Ins99]. Le dénominateur commun de ces variantes étant celle définie par Brian Kernighan et Dennis Richie en 1976. Chacune des variantes du langage C introduit ses propres extensions et sa propre sémantique. Bien que les spécifications du langage permettent l'élaboration d'un tisseur d'aspect statique par transformation du code source, ces informations ne suffisent pas à la conception d'un tisseur d'aspect à la volée tel qu'Arachne. Les nombreuses techniques de compilation et d'optimisation obscurcissent la relation entre le code source et le code machine. Par exemple, la réplication du code d'une fonction partout où celle-ci est appelée (technique appelée *inlining* de code) permet d'optimiser l'exécution du code en supprimant les instructions machine d'appel de routines et de sauvegarde de registres. De même, l'analyse des variables mortes (une variables est considérée morte à un moment de l'exécution lorsque son contenu est inutilisé ou sera écrasé quelle que soit la suite de l'exécution) permet la réutilisation des variables locales en vue d'économiser la mémoire. Ces deux techniques d'optimisation peuvent entraîner la suppression d'entités du code source (fonctions ou variables) dans le code machine. Ainsi, la compilation du

code n'est pas un procédé bijectif. De plus, les compilateurs C (Intel C compiler, GCC, *etc*) appliquent des patrons de code et d'affectation des registres divers. Les optimisations et le code spécifique à chaque compilateur rend difficile la compréhension du comportement d'un programme à la seule lecture de son code exécutable.

Il est illusoire de concevoir un tisseur d'aspect à la volée qui dépendrait d'un compilateur donné voire d'options de compilations spécifiques. De même, imposer l'utilisation d'une machine virtuelle ou d'un interpréteur source (par exemple CINT [DG87]) est incompatible avec les performances recherchées lors de la conception d'applications système. La construction d'un tisseur d'aspect à la volée pour le C doit donc trouver ses fondements sur les invariants de l'exécution d'un programme C. Les environnements d'exécution imposent une interopérabilité de fait : l'interface exposée par le système (appels systèmes, librairies...), le chargement des exécutables (le chargeur dynamique), l'éditeur de liens et le dévermineur, sont régis par plusieurs standards. Dans le cadre de ce document, nous nous sommes limités aux systèmes Linux sur plateformes Intel 32 bits. Cet environnement privilégie le format ELF pour la représentation sur disque et le chargement dynamique d'exécutables [Sta95], le DWARF définit le format de représentation des méta-informations permettant aux outils de débogage de rétablir la relation entre code machine et code source [Gro06], enfin l'*Application Binary Interface*, ou ABI, impose un modèle d'exécution des exécutables C [SU94]. Ces spécifications imposent leurs restrictions autour de trois axes : le format de représentation et la manipulation des données lors de l'exécution, les contraintes sur le code exécutable, et les interfaces proposées et les informations maintenues par l'environnement d'exécution.

Pour les données, l'ABI Linux définit la taille et la représentation en mémoire des types primitifs du langage C (caractères, entiers, réels, signés, non signés, courts, longs) et les contraintes d'alignements des variables et des membres des types construits. L'ABI contraint également les instructions machines devant être utilisées pour accéder aux données d'un programme, par exemple, l'accès à une variable locale doit se faire par des instructions machines à adressages relatifs, c'est-à-dire que l'adresse de l'accès est calculée par l'expression $base + offset$, où la base est l'adresse de la *frame* de la fonction courante. Les spécifications du langage C autorisent les compilateurs à ne pas mettre à jour le contenu des variables pendant les calculs dès lors que leur contenu est actualisé à la fin de chaque *statement* C (à chaque point-virgule dans le source).

Pour le code exécutable, l'ABI définit l'alignement des fonctions en mémoire et impose les instructions machines devant être utilisées pour appeler une fonction : un appel à une fonction publique se fait impérativement via l'instruction machine *call* avec un adressage direct, alors qu'un appel à une fonction d'une bibliothèque se fait en appelant une fonction tremplin qui contient un code résolu à l'exécution réalisant un saut vers la fonction de la bibliothèque. De même, l'accès à une variable globale doit toujours être réalisé par une instruction à adressage absolu.

Pour l'environnement d'exécution, l'ABI impose lors de l'exécution que le programme maintienne sur la pile une trace (*frame*) des fonctions en cours d'exécution. L'ABI décrit également le passage des arguments et l'appel en lui-même d'une fonctionnalité du système d'exploitation.

C'est sur ces standards que nous avons construit Arachne, afin que notre tisseur d'aspect soit indépendant du processus de compilation du programme de base.

$$
\begin{array}{lcl}
JP & ::= & \texttt{jp\_call}(\text{val fid}(\overrightarrow{val})) \\
 & | & \texttt{jp\_readGlobal}(\text{varId}, \text{val}) \\
 & | & \texttt{jp\_read}(@, \text{val}) \\
 & | & \texttt{jp\_writeGlobal}(\text{varId}, \text{val}, \text{size}) \\
 & | & \texttt{jp\_write}(@, \text{val}, \text{size}) \\
 & | & \texttt{jp\_cflow}(\overrightarrow{funId}, \text{cfEnd}) \\
 & | & \texttt{jp\_cflowstar}(\overrightarrow{funId}, \text{cfEnd}) \\
 & & \\
cfEnd & ::= & \texttt{jp\_call}(\text{val funId}(\overrightarrow{val})) \\
 & | & \texttt{jp\_readGlobal}(\text{varId}, \text{val}) \\
 & | & \texttt{jp\_writeGlobal}(\text{varId}, \text{val}, \text{size}) \\
 & & \\
Seq & ::= & \overrightarrow{JP} \\
 & & \\
val & ::= & \texttt{0 | 1 | 2 | ...} \quad \texttt{// int} \\
 & | & \texttt{@0 | @1 | @2 | ...} \quad \texttt{// int*} \\
 & | & \texttt{...} \quad \texttt{// values of other C types}
\end{array}
$$

FIG. 4.1: Le modèle de points de jonction d'Arachne

4.1.2 Les points de jonction

Dans la section précédente, nous avons discuté des standards qui régissent le langage C. Ceux-ci déterminent la syntaxe et la sémantique du langage et de ses variantes (C ANSI, *etc*). Ces standards fixent également la représentation d'un exécutable C sur disque (ELF, COFF) et établissent l'environnement d'exécution d'un processus C (ABI, POSIX). Nous avons pu voir que le modèle d'exécution d'un programme C est défini sur trois axes : la représentation des données, l'historique d'exécution et la représentation du code en mémoire. Dans un langage d'aspect, le modèle de points de jonction définit les événements dans l'exécution du programme tissé auxquels le langage de coupe peut faire référence. La figure 4.1 présente sous la forme BNF, notre modèle de points de jonction. Dans cette partie, nous présentons ce modèle à travers les trois axes précédemment cités de l'exécution d'un programme C : historique, données et code.

4.1.2.1 Événements du code

Notre modèle définit le point de jonction *jp_call (r fid($\overrightarrow{val}$))*. Celui-ci correspond à un appel à la fonction *fid* dans le programme de base. Ce point de jonction requiert la signature de la fonction *fid* : $\overrightarrow{val}$ indique le type des argument passés lors de l'appel à *fid* et *r* est le type de la valeur de retour.

4.1.2.2 Manipulations de données

Nous définissons quatre points de jonction sur la manipulation des données durant l'exécution d'un programme C. Ces points de jonction correspondent tous à l'accès à une variable. Nous distinguons ces quatre points de jonction en 2 catégories : les écritures et les lectures, pour lesquelles nous distinguons également deux variantes selon que l'accès est direct ou via un alias (pointeur).

- les accès en lecture
  - Le point de jonction *jp_readglobal (vname, value)* correspond à l'accès en lecture à la variable *vname* dont la valeur est associée à *value*
  - Le point de jonction *jp_read (vname, value)* correspond à l'accès en lecture via pointeur à la variable *vname* dont la valeur est associée à *value*
- les accès en écriture
  - Le point de jonction *jp_writeglobal (vname, value, size)* correspond à l'écriture de la valeur *value*, de taille *size* dans la variable de nom *vname*
  - Le point de jonction *jp_write (vname, value, size)* correspond à l'écriture via pointeur de la valeur *value*, de taille *size* dans la variable de nom *vname*

Il peut sembler surprenant que nous distinguions dans notre modèle, les accès directs des accès via pointeurs. Cette séparation est motivée par les contraintes de performance. En effet, à l'exécution, l'arithmétique sur pointeurs est nettement plus coûteuse que l'accès à une variable globale (dans un ordre de grandeur de un pour mille, *cf* 6.2 tableau 6.1).

4.1.2.3 Inspection de l'historique

Notre modèle définit trois points de jonction relatifs à l'historique d'exécution d'un programme C. Le premier point de jonction *jp_cflow(fn)* teste le flot d'exécution du programme vis-à-vis de la pile d'appel *fn* : si les n dernières fonctions appelées correspondent à la suite d'appels *fn*. Le second point de jonction *jp_cflowstar(fn)* est une variante du point de jonction *jp_cflow(fn)*. Cette variante teste également la pile d'appels vis-à-vis de la suite de *fn* mais cette fois, les appels ne sont pas nécessairement consécutifs.

Cette distinction peut paraître surprenante, encore une fois elle est motivée par les contraintes de performance. En effet, le temps nécessaire à la vérification d'un point de jonction *jp_cflow(fn)* dépend directement de n, tandis que dans le cas de *jp_cflowstar* ceci dépend du nombre de *frame* sur la pile.

Le troisième point de jonction relatif à l'historique d'exécution du programme de base est *Seq*. Il est déclenché par l'exécution ordonnée des points de jonction $\overrightarrow{JP}$. Contrairement, aux *jp_cflow* et *jp_cflowstar*, il n'est pas nécessaire que les points de jonction $\overrightarrow{JP}$ soient des points de jonction « appel de fonction », ni que leurs exécutions soient imbriquées. Seul l'ordre dans lequel les points de jonction $\overrightarrow{JP}$ s'exécutent importe.

4.2 Le langage de coupe

Un langage de coupe définit les constructions qui correspondent à l'apparition de points de jonction dans l'exécution du programme de base. Une coupe permet de rassembler sous la même dénomination un ensemble de points de jonction. Ainsi, le langage de coupe d'Arachne est construit au-dessus des abstractions définies dans notre modèle de points de jonction. Dans cette partie, nous présentons notre langage de coupe, ensuite nous introduisons les variables de coupe qui permettent d'associer des informations dynamiques liées aux coupes pour réaliser des tests dynamiques.

4.2.1 Sélecteurs de points de jonction

La figure 4.2 présente notre langage de coupes. Les coupes notées *PPrim*, peuvent être de trois types : appels de fonction, accès à une variable, ou flot de contrôle. Les coupes peuvent être combinées en utilisant un *ou* logique dénoté par ||.

Une coupe de type appels de fonction regroupe les points de jonction *jp_call (r f(argn))*, c'est-à-dire tous les appels à une fonction ayant pour signature *r f(argn)*. Les arguments de cette fonction peuvent être associés à des variables de coupe lors de la déclaration de la signature de *f*. De même, on peut également associer à une variable de coupe la valeur de retour de la fonction grâce à la construction *return (pattern)*.

Une coupe de type accès à une variable globale est noté *PAccGlobal*, elle correspond aux points de jonction soit en lecture *jp_readglobal (vname, value)*, soit en écriture *jp_writeglobal (vname, value)* (de manière exclusive) à la variable globale *vname* du programme de base. La variable lue, *respectivement écrite*, peut être associée à une variable de coupe grâce à la construction *value(pattern)* ; de plus, la taille lue (au sens C du terme), *respectivement écrite*, peut-être mémorisée dans une variable de coupe en utilisant la construction *size(varName)*. De même que pour les coupes sur les appels de fonctions, le pattern matching s'applique également aux constructions *value* et *size*.

Les coupes de type accès via pointeur, noté *PAcc*, vérifie les points de jonction de type *jp_read (vaddr, value)* et *jp_write (vaddr, value)*. Ces coupes sont symétriques à celles sur les accès à des variables globales.

Une coupe sur flot de contrôle (*PCf*) noté *controlflow(PCallSig, ... PCallSign), PCfEnd)* où *PCfEnd* est une coupe de type *PCall* ou *PAccGlobal*, dénote les points de jonction de type *jp_cflow(...)*. Il correspond à l'ensemble des points de jonctions relatifs à la coupe *PCfEnd* lorsque la pile d'appel du flot de contrôle du programme de base équivaut à la pile d'appels *PCallSig1, ..., PCallSign*. Les coupes de type flot de contrôle non stricts vérifient la pile d'appels *PCallSig1, ..., PCallSign* sans que ceux-ci soient directement imbriqués.

4.2.2 Variables de coupe

Nous avons vu qu'il était possible, par exemple dans la signature d'une fonction ou à l'aide de la construction *return*, de déclarer des variables de coupe. Le contenu de ces variables est automatiquement renseigné lors de l'exécution de la coupe. Une variable de coupe peut être utilisée dans une construction *if (cond)*, afin de réaliser des tests complémentaires lors du déclenchement de la coupe et dans l'action de l'aspect. Vis-à-vis du code des aspects, la portée d'une variable de coupe est limitée à l'aspect dans lequel elle est déclarée, à noter que dans le cas d'un aspect de type séquence, les variables de coupe sont accessibles à toutes les coupes et actions.

En plus des variables de coupe définies dans les signatures des points de jonction, la construction *bind (t vname, expr)*, permet de déclarer une variable de coupe *vname* de type *t*. À chaque fois qu'une coupe contenant un *bind* est déclenchée, le contenu de la variable de coupe *vname* est écrasée par le résultat de l'évaluation de l'expression C *expr*. L'expression *expr* n'est pas limitée aux informations contenues dans la coupe et peut donc utiliser des variables et des fonctions du programme de base. Au sein d'une séquence d'aspects, la construction *bind* permet ainsi de mémoriser une partie de l'état du programme de base lors du déclenchement d'une coupe et de l'utiliser dans

$$
\begin{array}{lcl}
PPrim & ::= & PCall \\
 & | & PAccGlobal \\
 & | & PCf \\
 & | & PPrim \;||\; PPrim \\
 & & \\
PCall & ::= & [\ PCallModifier\ ]\ PCallSig\ [\ \&\&\ \texttt{args(}\ \overrightarrow{pattern}\ \texttt{)}\ ]\ [\ \&\&\ \texttt{return(}\ \text{pattern}\ \texttt{)}\ ] \\
 & & [\ \&\&\ \texttt{bind(}\ \text{t vname, expr}\ \texttt{)}\ ]\ [\ \&\&\ PIf\ ] \\
PCallSig & ::= & \texttt{call(}\ \text{type funId(}\overrightarrow{type}\text{)}\ \texttt{)} \\
 & & \\
PIf & ::= & \texttt{if(}\ \text{expr}\ \texttt{)}\ \ [\ \&\&\ PIf\ ] \\
 & & \\
PAccGlobal & ::= & \texttt{readGlobal(}\ \text{type varId}\ \texttt{)}\ [\ \&\&\ \texttt{value(}\ \text{pattern}\ \texttt{)}\ ]\ [\ \&\&\ PIf\ ] \\
 & | & \texttt{writeGlobal(}\ \text{type varId}\ \texttt{)}\ [\ \&\&\ \texttt{value(}\ \text{pattern}\ \texttt{)}\ ] \\
 & & [\ \&\&\ \texttt{size(}\ \text{pattern}\ \texttt{)}\ ]\ [\ \&\&\ \texttt{bind(}\ \text{t vname, expr}\ \texttt{)}\ ]\ [\ \&\&\ PIf\ ] \\
 & & \\
PCf & ::= & \texttt{cflow(}\ PCallSigList,\ PCfEnd\ \texttt{)} \\
 & | & \texttt{cflowstar(}\ PCallSigList,\ PCfEnd\ \texttt{)} \\
 & & \\
PCallSigList & ::= & PCallSig\ \ [\ ,\ PCallSigList\ ] \\
 & & \\
PCfEnd & ::= & PCall \mid PAccGlobal \\
 & & \\
PAcc & ::= & \texttt{read(}\ \text{var}\ \texttt{)}\ [\ \&\&\ PIf\ ] \\
 & | & \texttt{write(}\ \text{var}\ \texttt{)}\ [\ \&\&\ \texttt{size(}\ \text{pattern}\ \texttt{)}\ ]\ [\ \&\&\ PIf\ ] \\
 & & \\
PCallModifier & ::= & \texttt{before} \\
 & | & \texttt{after} \\
 & & \\
pattern & ::= & var \mid val
\end{array}
$$

FIG. 4.2: Le langage de coupe d'Arachne

une autre coupe. Nous illustrerons dans l'exemple de la section 4.4.5 l'utilité de cette construction.

4.3 Le langage d'aspect

Nous venons d'introduire le langage de coupe d'Arachne. C'est au-dessus de ce langage de coupe que nous avons construit notre langage d'aspect. Nous présentons ici la grammaire de ce langage, avant de l'illustrer à travers des exemples.

4.3.1 Aspects

La figure 4.3 présente le langage d'aspect que nous avons conçu. Notre langage distingue les aspects en deux catégories : les aspects dits primitifs notés *AspPrim*, ou les séquences d'aspects primitifs notés *AspSeq*.

Les aspects primitifs associent une coupe primitive à une action qui sera exécutée à tous les points jonction correspondant à la coupe primitive. Lorsque la coupe est formée d'un « ou » logique de deux coupes, toute variable de coupe utilisée dans un test dynamique de l'aspect doit avoir été associée dans les deux membres du « ou ».

L'action notée *Advice* est un bloc d'instructions C qui est exécutée en lieu et place du point de jonction ayant déclenché l'aspect. L'action doit avoir la même signature que le point de jonction qu'elle remplace : c'est-à-dire, le même type de retour dans le cas d'un point de jonction de type appel de fonction, le même type que la variable lue dans le cas d'un accès en écriture à une variable, et du type C *void* dans le cas d'un accès en lecture.

Lorsque l'action n'est pas définie, le point de jonction ayant déclenché l'aspect est exécuté. De même, si l'action est définie par un bloc vide, le point de jonction déclenchant l'action ne sera pas exécuté. Les coupes sur les appels de fonction peuvent être préfixées par *before* ou *after* pour indiquer que l'action doit être exécutée avant ou après l'exécution du point de jonction.

Les aspects de type séquence, sont composés d'une séquence d'aspects primitifs. À chaque fois que le premier aspect d'une séquence est exécuté, ceci crée une nouvelle séquence en cours. Une séquence en cours progressera lorsque le deuxième aspect de celle-ci sera exécuté et ainsi de suite jusqu'au dernier aspect. Les aspects primitifs de la séquence à l'exception du premier et du dernier, peuvent être exécutés zéro ou plusieurs fois en utilisant la notation « * » sur cet aspect. Dans ce cas, cet aspect pourra être exécuté jusqu'à ce que l'aspect qui le suit dans la séquence, soit exécuté.

La construction *aspect until (as)* permet de désactiver l'aspect *aspect* dès que l'aspect primitif *as* sera exécuté.

Le langage d'aspect d'Arachne autorise les développeurs d'aspect à réutiliser dans les actions des aspects du code défini dans le programme de base. Par exemple, il est possible dans une action d'appeler une fonction du programme de base. Pour cela, le langage requiert que le développeur ait préalablement importé cette fonction en utilisant la construction *require Id as Type*, où *Id* est le nom de la fonction et *Type* est sa signature. Il existe un cas particulier pour les aspects sur les appels de fonction : une fonction *proceed* est implicitement déclarée, elle correspond à un alias de la fonction interceptée et peut être utilisée dans l'action.

4.3.2 Placement des aspects

Lors du tissage, l'utilisateur d'Arachne indique au tisseur d'aspect l'identifiant de processus du programme auquel tisser les aspects. Par défaut, tous les aspects seront tissés dans cette application. Il est néanmoins possible d'écrire un fichier contenant plusieurs aspects, lesquels seront tissés sur des applications différentes. Pour cela, il est nécessaire de préfixer la définition des aspects par un identifiant de placement. Cet identifiant peut correspondre à une ou plusieurs applications. C'est lors du tissage que l'utilisateur devra indiquer pour chaque identifiant de placement le ou les identifiants de processus correspondants. Par défaut, il existe un identifiant de placement prédéfini « K » désignant le noyau Linux.

Le langage d'aspects d'Arachne autorise non seulement l'écriture d'aspects tissés sur plusieurs applications mais également des aspects partagés sur plusieurs applications et/ou le système. Ainsi, un aspect de type séquence peut être défini de sorte que les étapes la composant soient dispersées sur plusieurs applications et/ou le noyau (pour cela, chaque étape doit être préfixée d'un indicateur de placement). Comme les étapes d'une séquence peuvent utiliser mutuellement leur données, celles-ci doivent être accessibles à toutes parties de l'aspect. Ainsi, lorsque un aspect est partagé par

$$
\begin{array}{lcl}
\mathit{Asp} & ::= & \mathit{RequireStmt}\ \ \mathit{Asp} \\
 & | & \mathit{AspPrim}\ \ [\ \&\&\ \texttt{until(}\ \mathit{AspPrim}\ \texttt{)}\ ] \\
 & | & \mathit{AspeSeq}\ \ [\ \&\&\ \texttt{until(}\ \mathit{AspPrim}\ \texttt{)}\ ] \\
 & & \\
\mathit{RequireStmt} & ::= & \texttt{require}\ \ \mathit{Id}\ \ \texttt{as}\ \ \mathit{Type}\ \texttt{;} \\
 & & \\
\mathit{AspPrim} & ::= & \mathit{PPrim}\ \ \mathit{Advice} \\
 & & \\
\mathit{AspSeq} & ::= & \texttt{seq(}\ \ \mathit{AspPrim} \\
 & & \qquad \mathit{AspSeqElts} \\
 & & \qquad \mathit{AspSeqElt}\ \texttt{)} \\
 & & \\
\mathit{AspSeqElts} & ::= & \mathit{AspSeqElt}\ \ [\mathit{AspSeqElts}] \\
 & | & \mathit{AspSeqElt}\ *\ \ [\mathit{AspSeqElts}] \\
 & & \\
\mathit{AspSeqElt} & ::= & \mathit{AspPrim} \\
 & | & \mathit{PAcc}\ \mathit{Advice} \\
 & | & (\mathit{AspSeqElt}\ ||\ \mathit{AspSeqElt}) \\
 & & \\
\mathit{Advice} & ::= & \texttt{;} \\
 & | & \texttt{then}\ \mathit{funId}(\overrightarrow{\mathit{pattern}})\ \texttt{;} \\
 & & \\
\mathit{pattern} & ::= & \mathit{var} \\
 & | & \mathit{value}
\end{array}
$$

FIG. 4.3: Le langage d'aspect d'Arachne

plusieurs applications et/ou le noyau, Arachne met en place une mémoire partagée accessible à toutes les applications tissées sur laquelle les données des aspects sont stockées.

4.4 Exemples illustrateurs

Nous venons d'introduire le langage d'aspect d'Arachne qui associe un langage de coupe à des actions écrites en C. Le langage de coupe sélectionne les événements du programme déclenchant les actions et permet de mémoriser des données relatives aux événements interceptés qui peuvent être utilisées dans l'action de l'aspect ou dans le langage de coupe pour conditionner le déclenchement des actions.

Le langage d'Arachne regroupant un nombre conséquent de notions, nous allons maintenant présenter ce langage en *action* à travers les exemples présentés dans le chapitre 3. Ces scénarios d'utilisation nous ont également servi à l'évaluation de notre prototype, ils seront réutilisés dans le chapitre 6. Afin de faciliter la compréhension de notre langage, nous revisitons nos exemples dans un ordre différent de celui dans lequel ils ont été présentés. À chaque exemple, nous introduirons un nouvel aspect du langage d'Arachne. D'abord, nous montrerons comment empêcher l'exploitation de l'allocateur de mémoire de la librairie standard C à des fins malveillantes. Dans cet exemple, nous introduisons les éléments les plus simples du langage d'Arachne : les aspects interceptant les appels de fonctions. Ensuite, nous présentons les aspects ajoutant une politique de préchargement dans le cache Web Squid. Cette fois, nous présentons les aspects sur

```
extern void add_to_set (set_t, set_t);
extern void remove_from_set (set_t, set_t);
extern int exists (set_t, set_t);

set_t allocated, freed;

call (void* malloc (size_t size)) && return (buffer)
then {
    remove_from_set (freed, buffer);
    add_to_set (allocated, buffer);
}

call (void free (void* buffer))
then {
    if (exists (freed, buffer)) error ();
    else if (!exists (allocated, buffer)) warning ();

    free (buffer);
    remove_from_set (allocated, buffer);
    add_to_set (freed, buffer);
}
```

Listing 4.1: Un aspect pour détecter les « double free bugs »

les flots de contrôle qui étendent naturellement les aspects sur les appels de fonctions. Des aspects sur les flots de contrôle, nous passerons à la deuxième forme de manipulation de l'historique de l'exécution que sont les aspects de type séquence à travers le remplacement du protocole TCP par UDP dans un programme client. Ensuite, nous verrons comment protéger une application de l'exploitation des débordements de tableaux. Toujours autour d'une séquence, cette fois nous introduisons l'interception des accès aux variables. Nous présentons un aspect détectant les inter-blocages de verrous, ce qui nous permettra de couvrir la construction *bind*. Enfin, nous terminons par la supervision de l'exécution du cache Web Squid pour laquelle nous utilisons la fonctionnalité de placement des aspects.

4.4.1 Concepts de base – Exploitation de l'allocateur de mémoire C

Nous reprenons ici l'exemple de la section 3.1.1. Le listing 4.1 montre un fichier d'aspect permettant de détecter les erreurs d'allocation mémoire dans un programme C. Pour cela, nous utilisons des ensembles (ligne 5) représentant les zones mémoire allouées et celles libres. Pour manipuler ces ensembles, nous définisons trois fonctions *add_to_set*, *remove_from_set* et *exists* pour respectivement ajouter, retirer à un ensemble et tester l'appartenance d'un ensemble dans un autre. Nous définissons un aspect pour intercepter les réservations de mémoire (appels à *malloc*, ligne 7). Lors de cet appel, la coupe récupère l'adresse de la zone mémoire allouée et l'action de l'aspect met à jour les deux ensembles définis précédemment. Un deuxième aspect (ligne 13) intercepte la libération d'une zone mémoire afin de maintenir les deux ensembles et de valider l'intégrité de la requête (ligne 15), c'est-à-dire vérifier que la zone est bien allouée avant de la libérer.

4.4.2 Les flots de contrôle – Préchargement dans le cache Web Squid

Nous reprenons ici l'exemple de la section 3.2. Pour ajouter une politique de préchargement dans Squid, nous allons brancher sur l'exécution de Squid des aspects qui

```
require Number_of_Fd as int*;
require Squid_Max_Fd as int*;

controlflow (
    void clientSendMoreData (void*, char*, size_t),
    call (HttpReply* clientBuildReply (clientHttpRequest* request,
        char* buffer,
        size_t bufferSize))
then {startPrefetching(request, buffer, bufferSize);}
&& until (writeGlobal (int* Number_of_Fd)
    && if ((*Number_of_Fd) * 100 / (*Squid_Max_Fd) > 74)))

controlflow (
    void clientSendMoreData (void*, char*, size_t),
    call (void comm_write_mbuf (int fd, MemBuf mb, void* handler, void* handlerData))
        && if (! isPrefetch (handler))
then {parseHyperlinks (fd, mb, handler, handlerData);}

call (void clientWriteComplete (int fd, char* buf, size_t size,
    int error, void* data))
    && if (! isPrefetch (handler)
then {retrieveHyperlinks (fd, mb, handler, handlerData);}
```

Listing 4.2: Un aspect pour ajouter le pré-chargement dans Squid

vont pour chaque requête Web d'un utilisateur du réseau, précharger les pages Web référencées par le résultat de la requête.

Les aspects du listing 4.2 ajoutent cette politique de préchargement dans le cache Web Squid. Ce listing débute par deux clauses *require* qui indiquent au tisseur d'aspect que *Number_of_Fd* et *Squid_Max_Fd* sont deux variables du programme de base. La politique est introduite grâce à trois aspects, deux portent sur des flots de contrôle (lignes 4 et 13), et le dernier sur un appel de fonction (ligne 19). Le premier flot de contrôle initialise le préchargement (ligne 9) lorsque Squid prépare une réponse (appel à *clientBuildReply*, ligne 6) à la requête d'un client (dans *clientSendMoreData*, ligne 5)). La clause *until* permet ici de désactiver le préchargement lorsque trop de connections réseaux sont ouvertes (75% du nombre maximum autorisé dans Squid, ligne 10). Le deuxième aspect de type flot de contrôle déclenche la recherche des hyperliens d'une page reçue (ligne 17)(c'est-à-dire lors d'un appel à *comm_write_mbuf*, ligne 15, dans la fonction *clientSendMoreData*, ligne 14). Finalement, le dernier aspect va précharger les pages pointées par les hyperliens récupérées par le deuxième aspect. Pour cela, l'aspect remplace les appels à la fonction *clientWriteComplete* par un appel à *retrieveHyperlinks* (ligne 19).

4.4.3 Les séquences – Remplacement de TCP par UDP

Nous présentons ici une solution à la problématique illustrée dans la section 3.3.2. L'aspect du listing 4.3 permet de changer le protocole de communication utilisé de TCP à UDP. Le mot-clé *seq* indique que l'aspect est une séquence. Cette séquence comporte quatre étapes. La première étape (ligne 2), c'est-à-dire l'initialisation du protocole TCP est un appel à la fonction *socket* avec trois arguments : *AF_INET*, *SOCK_STREAM*, et *0*. Seuls les appels à *socket* avec ces trois constantes sont interceptés. Lorsque la coupe est déclenchée, l'aspect remplace les arguments de l'appel par les constantes correspondant au protocole UDP : *AF_INET*, *SOCK_DGRAM*, et *0* (ligne 3). Lorsque l'appel à *socket* est terminé, le descripteur de fichier retourné est mémorisé par la construction *return* dans la variable de coupe *fd* (ligne 2). Dès lors, le prochain appel à la fonction *connect* sur ce descripteur de fichier déclenchera la coupe

```
aspect toUDP :: seq (
    call (int socket (int AF_INET, int SOCK_STREAM, int 0)) && return (fd)
        then {socket (AF_INET, SOCK_DGRAM, 0);}
    call (int connect (int fd, struct sockaddr* saddr, socklen_t slen))
        then {return 0;}
    (call (size_t read (int fd, void* rbuffer, size_t rlen))
        then {return recvfrom (fd, rbuffer, rlen);}
||  call (size_t write (int fd, void* wbuffer, size_t wlen))
        then {return sendto (fd, wbuffer, wlen, 0, saddr, slen);})*
    call (int close (int fd))
)
```

Listing 4.3: Un aspect pour passer de TCP à UDP

```
aspect bo :: seq (
    call (void* malloc (size_t s)) && return (buffer)
    write (buffer) && size (idx) && if (idx >= s)
        then {reportOverflow ();} *
    call (void free (void* buffer))
)
```

Listing 4.4: Un aspect pour détecter les débordements de tableaux

de l'étape 2 (ligne 4). Les autres arguments de *connect* : l'adresse d'une structure décrivant le paramétrage de la connection et la taille de cette structure, sont mémorisés dans les variables de coupe *saddr* et *slen* (ligne 4). UDP étant un protocole déconnecté, l'action de cette étape supprime l'appel à *connect* et renvoie la valeur zéro au programme de base (ligne 5). Une fois la phase d'initialisation terminée, les appels aux fonctions *read* et *write* sur le descripteur de fichier *fd* sont changés en leurs équivalents UDP : *recvfrom* et *sendto*. L'opérateur « * » indique que cette troisième étape de la séquence peut être déclenchée de zéro à une infinité de fois. Finalement, l'aspect attend un appel à *close* pour terminer l'instance de la séquence (ligne 10).

4.4.4 Les accès aux variables – Les débordements de tableaux

Nous présentons ici une solution à la problématique illustrée dans la section 3.1.2. L'aspect présenté dans le listing 4.4 permet de détecter les débordements de tableaux d'un programme C. Cet aspect est une séquence de trois étapes qui correspondent à l'utilisation des tableaux dynamiques en C. La première étape (ligne 2), détecte l'allocation dynamique du tableau via la fonction *malloc*. Lors de cet appel, la taille de la mémoire allouée et l'adresse de base du tableau, c'est-à-dire l'argument, respectivement la valeur de retour de *malloc* sont mémorisées dans les variables de coupe, *s* et *buffer*. Cette première étape n'ayant pas d'action définie, aucune action n'est ajoutée à l'appel de *malloc*. Une fois l'allocation dynamique réalisée, tous les accès en écriture à ce tableau seront interceptés (ligne 3) jusqu'à ce que ce tableau soit désalloué par un appel à la fonction *free* ayant pour argument l'adresse de base du tableau (ligne 5). Pour intercepter les écritures dans le tableau (ligne 3), nous utilisons la construction *write* du langage sur l'adresse de base du tableau. La construction *size* permet alors de mémoriser dans la variable de coupe *idx*, l'index de l'accès. Cet index correspond au décalage en octets par rapport à *buffer*. Si ce décalage est supérieur à la taille du tableau qui a été mémorisée dans la première étape de l'aspect, alors l'accès au tableau est remplacé par une alarme (ligne 4).

```
extern int isDeadLocked();

aspect detection :: seq(
    call(lock_id create_lock() && return(lid1));

    (   before call(void require(lock_id lid2)
        && bind(tid_t tid, gettid()) && if (lid1==lid2))
                then isDeadLocked() ? alarm() : 0;

        after call(void require(lock_id lid3)
        && if (tid == gettid() && lid1==lid3))
            then isDeadLocked() ? alarm() : 0;
        )
    ||
        call(void release(lock_id lid4))
        && if (tid == gettid() && lid1==lid4));
    ) *

    call(void destroy_lock(lock_id lid4)
    && if (lid1==lid4));
)
```

Listing 4.5: Un aspect pour détecter les « deadlocks »

4.4.5 *bind* – Les interblocages de verrous

Nous reprenons ici l'exemple de la section 3.1.3. Pour corriger ce problème, notre solution utilise un graphe des ressources allouées pour suivre les réservations des verrous. L'aspect présenté dans le listing 4.5 permet de détecter les inter-blocages de verrous. Pour simplifier le code source, nous ne présentons pas la fonction *isDeadLocked* qui parcourt le graphe à la recherche de cycles (ligne 1). L'aspect présenté, est construit sur une séquence afin de faire apparaître le protocole d'utilisation des verrous dans un programme C. Les trois étapes de la séquence correspondent respectivement à la création, l'utilisation et la destruction des verrous (lignes 4,6 et 19). La deuxième étape de la séquence supervise l'utilisation des verrous. Cette étape est formée d'un « ou » logique (ligne 14). Cette disjonction va intercepter la prise (ligne 6) ou la libération (ligne 15) d'un verrou.

Le premier membre de cette disjonction présente une particularité : il est constitué d'un double aspect. Ceci n'est autorisé que dans un cas bien précis : uniquement si les deux aspects portent sur le même point de jonction, c'est-à-dire quand le déclenchement du premier entraîne de manière certaine le déclenchement du second. Ce double aspect permet de distinguer la prise d'un verrou qui est faite via l'appel à *require* (ligne 6) de l'affectation en elle-même du verrou au demandeur qui est effective lors du retour de l'appel à *require* (ligne 10). Dans les deux cas : demande et affectation du verrou, les actions des aspects vérifient que la situation n'est pas bloquée. Lors de l'interception de la demande de réservation du verrou, nous utilisons la construction *bind* (ligne 7) afin de mémoriser l'identifiant du processus léger demandeur. En effet, il est nécessaire de vérifier que lorsque ce verrou sera libéré par un appel à *release* (ligne 15), l'appelant est bien le processus léger possédant ce verrou (ligne 16). Finalement, la séquence s'arrête lors de la destruction du verrou (ligne 19).

```
aspect monitoring :: seq (
    A:  call (void clientProcessRequest (struct clientHttpRequest* http)
        && bind (pid, GET_PID) && bind (IP_CLIENT, http->request->client_addr))
4
    K:  call (ssize_t vfs_read (struct file* file,
        char* buf, size_t count, loff_t* pos)
        && if (!isSocket (file) && !isPipe (file) && (current->pid == pid)))
        then {addClientReadDiskAccess (pid, IP_CLIENT, size);
9       return proceed (file, buf, count, pos);}
)
```

Listing 4.6: Un aspect pour superviser l'utilisation par client de l'espace disque dans Squid

4.4.6 Placement des aspects – La supervision du cache Web Squid

Nous reprenons ici l'exemple de la section 3.3.1. L'aspect du listing 4.6 réalise la supervision de l'utilisation en lecture des disques[1]. Cet aspect est construit autour d'une séquence de deux aspects primitifs. Le premier est placé dans Squid (ligne 2), tandis que le second est placé dans le noyau du système d'exploitation (ligne 5). La première étape de cette séquence intercepte les appels à la fonction *clientProcessRequest*, fonction appelée pour toute requête HTTP traversant le cache Web Squid. Cet aspect ne modifie pas l'appel intercepté mais se contente de mémoriser l'identifiant de processus associé avec l'instance de Squid dans lequel *clientProcessRequest* est appelé ainsi que l'adresse IP de laquelle émane la requête.

La deuxième partie de la séquence (ligne 5), celle se trouvant dans le noyau du système d'exploitation, intercepte les appels à la fonction *vfs_read*. Cet appel correspond à la lecture consécutive à l'appel à *clientProcessRequest* dans Squid. Lors de l'appel, la coupe teste l'argument *file* (le fichier dans lequel est réalisé la lecture) pour vérifier que celui-ci n'est pas un fichier spécial (*pipeline*, *socket*) (ligne 7). S'il ne l'est pas, on journalise cet accès via la fonction *addClientReadDiskAccess* (ligne 8). Dans tous les cas, on procède à l'appel originalement intercepté. Pour journaliser les lectures, notre fonction *addClientReadDiskAccess* entretient une liste associant adresses IP et quantité de données lues.

4.5 Conclusion

Dans ce chapitre, nous avons présenté le langage d'aspect d'Arachne. Nous avons conçu ce langage avec pour objectif de rester proche des concepts du langage C. Ainsi, notre langage permet d'écrire des aspects raisonnant sur le déclenchement des fonctions, la manipulation des données et l'historique d'exécution d'un programme. Après avoir justifié le choix du modèle de points de jonction de notre langage vis-à-vis du modèle d'exécution des programmes C, nous avons présenté notre langage de coupe et notre langage d'aspect.

Nous avons également souhaité illustrer notre langage *en action* à travers des scénarios d'utilisation mettant graduellement en avant l'expressivité de notre langage. Ces scénarios concernent des problématiques réelles, comme la sécurité des systèmes informatique, et ont fait l'objet d'évaluations en production dont une partie est présentée dans le chapitre 6.

[1] On présente ici l'utilisation en lecture, celle en écriture reste similaire mais pour des raisons de simplicité et de lisibilité on présente l'utilisation en lecture.

To program is to understand.

Kristen Nygaard

Chapitre 5

Arachne – une mise en œuvre modulaire

Où nous présentons l'implémentation modulaire d'Arachne, notre système de tissage dynamique par réécriture du code exécuté par le processeur. Les techniques de réécriture utilisées dans Arachne diffèrent de celles des outils existants. L'utilisation de la sémantique du code assembleur permet à Arachne de déclencher les aspects plus rapidement que les outils comparables. De plus, la conception modulaire d'Arachne, nous a permis de créer trois extensions du système de tissage dont une pour le tissage d'aspect dans le noyau Linux.

Sommaire

5.1 Vue d'ensemble **60**
5.1.1 Le compilateur d'aspect 61
5.1.2 Le tisseur d'aspect 61
5.1.3 Organisation fonctionnelle 62
5.1.4 Extensibilité 62
5.2 Injection du noyau de réécriture **63**
5.3 Interception du flot d'exécution **64**
5.3.1 Technique de réécriture des points de jonction 64
5.3.2 Les crochets – cohérence du déclenchement des aspects . . . 66
5.4 Compilation des aspects **67**
5.4.1 Le code exécutable des aspects 68
5.4.2 Les directives de tissage 70
5.5 Résolution des informations de tissage **71**
5.5.1 Résolution des symboles 72
5.5.2 Localisation des points de jonction 73
5.6 Conclusion **74**

Dans le chapitre précédent, nous avons décrit le langage d'aspect d'Arachne. Lors de sa conception, nous avons extrait des spécifications du langage C, les événements d'exécution d'un programme C pouvant faire l'objet d'une interception par des aspects. Notre langage exprime des actions réalisées en réponse à des événements de l'exécution du programme de base. Les événements de base reconnus par le langage sont les appels de fonctions et les accès aux variables. Notre langage permet également de raisonner sur l'historique du programme en capturant l'apparition

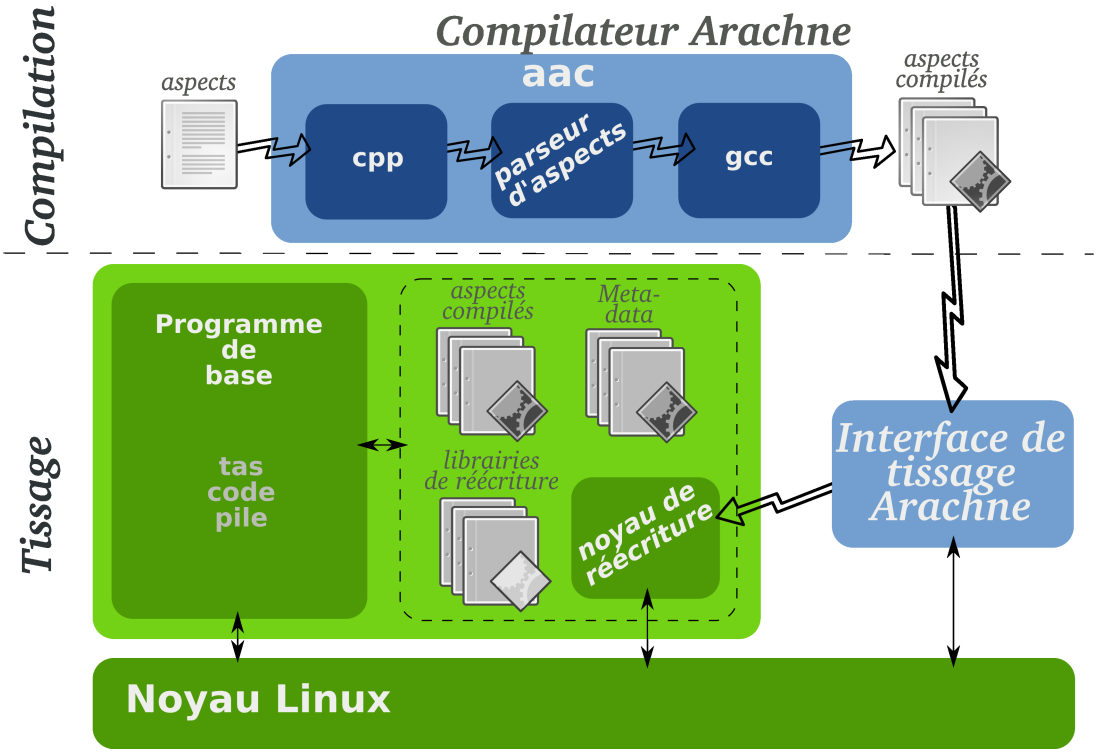


FIG. 5.1: Architecture générale du système Arachne

séquentielle ou imbriquée d'événements basiques.

Dans ce chapitre, nous allons présenter l'implémentation du système Arachne. Pour commencer, nous donnons une vue d'ensemble du système englobant chaque étape de l'écriture de l'aspect jusqu'à son exécution au sein de l'application tissée. Ensuite, nous détaillerons chacune des étapes du processus de tissage. D'abord nous discuterons de la compilation des aspects, puis de leur chargement dans l'environnement d'exécution du code à tisser. Ensuite, nous détaillerons comment à partir des coupes des aspects, sont localisés les points d'interactions des aspects avec le code à tisser. Nous montrerons comment est réalisé le détournement du flot d'exécution au niveau des points de jonction vers le code des aspects. Nous verrons également que son organisation modulaire facilite le développement d'extension du système de tissage.

5.1 Vue d'ensemble

Avant de rentrer en détail dans l'architecture et les mécanismes d'Arachne, nous présentons ici une vue d'ensemble du système. Cette vue d'ensemble précise les différentes phases du cycle de vie d'un aspect et présente l'organisation interne du système Arachne, c'est-à-dire quelles sont les responsabilités de chaque entité dans le processus de tissage. La figure 5.1 illustre l'organisation d'Arachne. Du point de vue de l'utilisateur, le système Arachne est composé de deux entités, le compilateur d'aspects et le tisseur d'aspects.

5.1.1 Le compilateur d'aspect

Le compilateur d'aspect est un outil en ligne de commande nommé *aac*, pour Arachne Aspect Compiler. Il est chargé de transformer un fichier d'aspects en fichiers C. Ces fichiers sont ensuite compilés avec *gcc* en un ensemble de librairies partagées (les « .so » sur les plateformes UNIX). Les fichiers d'aspects compilés contiennent le code exécutable correspondant aux actions des aspects ainsi qu'aux tests dynamiques des coupes. Ils embarquent également pour chaque aspect des directives de tissage. Ces directives sont des descriptions normalisées des coupes des aspects dont les points de jonction seront résolus lors du tissage. Comme les aspects compilés contiennent les directives de tissage, le fichier source des aspects n'est plus nécessaire lors du tissage. Les directives de tissage ne sont pas relatives à une application cible : les références au programme de base ne sont pas codées en dur mais utilisent les noms des symboles du programme. Ainsi, les aspects compilés sont indépendants du programme à tisser. Plus précisément, un aspect compilé peut être tissé sur n'importe quelle application dès lors que celle-ci contient les symboles référencés dans les directives de tissage.

L'outil *aac* utilisant le compilateur *gcc*, l'utilisateur peut par la ligne de commande définir des options propres à *gcc*. Par exemple, il est possible d'utiliser l'option *-I* pour définir des macros du préprocesseur C, ou de lier les aspects à des librairies partagées grâce à l'option *-l*. En fait, toute option passée lors de l'invocation à *aac* et qui ne lui est pas propre est transférée à *gcc* lors de la compilation du code C généré.

5.1.2 Le tisseur d'aspect

Le tisseur charge les aspects compilés dans le programme de base et établit les liens entre les aspects et le programme de base. Le tisseur d'aspect peut ainsi être vu comme un éditeur de liens dynamique. De même que le compilateur d'aspect, le tisseur est un outil en ligne de commande appelé *weaver*. Bien que la commande *weaver* apparaisse comme réalisant le tissage, c'est en fait une interface (un *frontend*) ayant trois fonctions. La première est de charger le noyau de réécriture dans les applications à tisser. C'est le noyau de réécriture qui réalise le tissage à proprement parler. La seconde est de transférer les requêtes de tissage et de détissage passées par l'utilisateur en paramètre de la ligne de commande au noyau de réécriture. Enfin, la troisième et dernière fonction de *weaver* est de fournir au noyau de réécriture les informations nécessaires au tissage des aspects. C'est *weaver* qui à la demande du noyau de réécriture, localise les points de jonction.

L'utilisation du *frontend weaver* est plutôt simple. Cette commande nécessite trois arguments. Le premier, passé par l'option *–pid* correspond au processus auquel appliquer la requête indiquée par l'argument *–request*. La requête peut être soit *WEAVE* soit *UNWEAVE* pour tisser et respectivement détisser le fichier d'aspect passé par l'argument *-f*. Un dernier argument optionnel *-d* permet de spécifier le niveau des messages de débogage par une chaîne de caractères spécifiant les types de messages désirés (informatifs, erreurs, relatifs au tissage, *etc*). Pour tisser un fichier d'aspects placés sur plusieurs applications, l'utilisateur doit affecter un ou plusieurs identifiants de processus pour chaque groupe défini dans le code des aspects. Pour ce faire, l'utilisateur fournit à l'option *–pid* une liste associative d'identifiants groupe/processus.

5.1.3 Organisation fonctionnelle

Nous venons de décrire les deux entités mises à disposition de l'utilisateur d'Arachne pour programmer par aspect : le compilateur d'aspect *aac* et le tisseur d'aspect *weaver*. Nous avons vu que ces outils étaient avant tout des interfaces (*frontend*). En effet, le système Arachne est architecturé autour de trois entités : les deux outils vus précédemment *aac* et *weaver* ainsi que le noyau de réécriture.

Après la compilation d'un fichier d'aspect grâce à *aac*, le déploiement des aspects compilés est réalisé à la demande de l'utilisateur par la commande *weaver*. Cette commande est un *frontend* relayant les requêtes de tissage au noyau de réécriture préalablement chargé dans le*(s)* espace*(s)* d'adressage*(s)* des programmes tissés.

Lors de la réception d'une requête, le noyau de réécriture charge dynamiquement les librairies de réécriture correspondant aux aspects à tisser et instrumente le programme en conséquence. Pour cela, les librairies de réécriture interrogent le *frontend* de tissage qui leur fournit les informations nécessaires à la réécriture du code, c'est-à-dire la localisation des points de jonction. Ainsi, les mécanismes de réécriture implémentées dans les librairies de tissage sont séparées du noyau de réécriture. Cela rend l'intégration de nouvelle librairies de tissage et l'extension du langage d'aspect plus aisées.

Cette approche présente d'autres avantages. Le noyau de tissage est indépendant des différents types d'aspects, du langage d'aspect et du programme de base. Les spécificités du langage d'aspects sont isolées dans le compilateur d'aspects. Alors que les spécificités liées à la plateforme d'exécution sont capturées par le noyau de réécriture et celles relatives à l'architecture processeur sont modularisées dans les librairies de réécriture. Ce découpage permet de limiter les traitements réalisés par le noyau de réécriture et donc de restreindre le code exécuté en cohabitation (dans le même espace d'adressage) avec le programme à tisser.

5.1.4 Extensibilité

Le système Arachne présente une architecture modulaire : la compilation des aspects est indépendante des techniques de réécriture qui sont implémentées dans des librairies chargées dynamiquement par le noyau de réécriture lors du tissage. Au cours de notre étude, nous avons pu éprouver l'extensibilité de ce système. Nous avons implémenté deux extensions du système Arachne. Historiquement, le tissage des aspects dans le noyau Linux est à l'origine une extension du système.

Nous avons également implémenté une librairie de réécriture et une librairie de résolution des points de jonction pour le langage C++. Ces librairies permettent de tisser des aspects sur les appels de méthodes des programmes C++. Cette implémentation n'est pas présentée dans cette thèse car elle n'a fait l'objet que de travaux préliminaires. Notamment, nous envisageons de concevoir un langage d'aspect spécifique au langage C++ et d'implémenter d'autres librairies de réécriture spécifiques au C++ en particulier pour le mécanisme d'héritage. En effet, contrairement au C, les exécutables C++ embarquent beaucoup d'informations sur les sources du programme, offrant ainsi des opportunités pour un langage d'aspect plus évolué.

Nous avons également implémenté une librairie de réécriture et une librairie de résolution des points de jonction pour l'interception des accès aux variables locales C. Cette extension utilise les informations de débogage au format DWARF2 pour

localiser les points de jonction et le mécanisme de réécriture par défaut utilisant les interruptions logicielles (section 5.3). Cette extension n'est également pas présentée dans ce document, car elle n'a fait l'objet d'aucune publication à l'heure actuelle.

5.2 Injection du noyau de réécriture

Le noyau de réécriture est responsable du chargement et du tissage des aspects en réponse aux requêtes de *weaver*. Avant tissage, le programme de base s'exécute normalement sans modification du langage de programmation utilisé, des sources ou de la chaîne de compilation. Or, le noyau de réécriture pour accéder à l'image mémoire du programme à tisser doit être chargé dynamiquement dans cet espace mémoire. Le noyau de réécriture permet le tissage d'aspect soit dans des applications en espace utilisateur, soit dans le noyau Linux.

Dans le cas du tissage dans le noyau Linux, le chargement du noyau de réécriture ne pose pas de complication : il est chargé sous la forme d'un module noyau. En revanche, pour les applications utilisateur, le noyau de réécriture est une librairie dynamique partagée. Or, il n'existe aucune méthode standard pour forcer une application à charger une librairie.

Pour cela, nous avons recourt à une solution *ad-hoc* : lorsque le *frontend weaver* reçoit une requête de tissage pour laquelle l'application à tisser ne contient pas de noyau de tissage, *weaver* va alors s'attacher à l'application à l'aide de l'appel système *ptrace* (*process trace*). Cet appel système permet à un processus, ici le *frontend weaver*, de contrôler l'exécution d'un autre processus, ici l'application à tisser. Le processus contrôleur peut, une fois attaché, intercepter les signaux émis vers le processus contrôlé et consulter et modifier l'espace mémoire et les registres du processus contrôlé.

Une fois *weaver* attaché au programme de base, il va sauvegarder les registres de l'application. Ensuite il va enregistrer le code de l'application à partir de l'endroit où celle-ci s'est arrêtée. Il remplace alors ce code par une fonction appelant l'éditeur de liens pour charger le noyau de réécriture et déclenchant un signal une fois le chargement terminé. Une fois le remplacement effectué, *weaver* redémarre l'exécution de l'application qui va exécuter le nouveau code et de ce fait, charger le noyau de réécriture. Une fois ce chargement terminé, le signal déclenché est intercepté par *weaver*. *weaver* remet alors le code et les registres de l'application dans leur état initial et redémarre l'application qui reprend son exécution là où elle s'était arrêtée avant l'intervention de *weaver*.

Dès lors, le noyau de réécriture est chargé, il crée un processus léger qui va ouvrir une socket réseau et attend les requêtes de tissage émises par *weaver*. Bien que l'utilisation de l'appel système *ptrace* permette la modification du code d'un programme, nous utilisons *ptrace* uniquement lors du chargement du noyau de réécriture. En effet, l'utilisation de *ptrace* stoppe l'exécution de l'application modifiée. Cela ne présente qu'un défaut mineur lors du chargement du noyau de réécriture pendant lequel l'application n'est interrompue qu'une centaine de milli-secondes, mais impliquerait des ralentissements importants s'il était utilisé pour le tissage d'aspects.

5.3 Interception du flot d'exécution

Comme nous le montre la figure 5.1, le tissage des aspects est réalisé par le noyau de réécriture depuis l'environnement d'exécution du programme de base, c'est-à-dire depuis l'espace mémoire de l'application ou depuis l'espace noyau. En effet, les spécifications du langage C n'interdisent pas le développement de programme auto-modifiant : le C autorise les accès mémoire à des adresses arbitraires fixées par le programmeur. De ce fait, rien n'empêche un programme de consulter ou de modifier son propre code. Néanmoins pour des questions de sécurité, les architectures modernes disposent d'un mécanisme de protection des pages mémoire. Ainsi, la plupart des systèmes d'exploitation interdisent par défaut qu'une page soit à la fois exécutable et modifiable. Ceci empêche la modification des pages mémoire contenant le code exécutable d'une application ou l'exécution de code stocké dans des pages contenant des données. Pour contourner cette restriction le noyau de réécriture utilise l'appel système *mprotect* pour modifier les protections des pages mémoire.

La modification du code exécutable d'une application est une condition indispensable mais insuffisante pour le tissage d'aspect. Le code tissé et le noyau de tissage s'exécutant de manière concurrente dans le même environnement d'exécution, deux problèmes se posent alors pour garantir la cohérence de l'exécution. Le premier est de garantir l'atomicité du tissage, c'est-à-dire qu'aucun aspect ne doit s'exécuter avant que tous les points de jonction de tous les aspects ne soient réécrits. Le second est de garantir la cohérence de l'exécution du programme de base pendant le tissage. Nous présentons ici les solutions mises en place pour résoudre ces deux problèmes. D'abord, nous verrons comment la réécriture des points de jonction garantit la cohérence de l'exécution du programme de base. Puis, nous présenterons le système des crochets qui garantissent l'atomicité du tissage.

5.3.1 Technique de réécriture des points de jonction

Lors de la réécriture des points de jonction, le noyau de tissage doit modifier un ensemble de points dans le code exécutable pour y rediriger l'exécution du code vers les aspects. Or, les processeurs Intel 32 bits ne disposent que de trois mécanismes autorisant le transfert de l'exécution : les branchements, les interruptions et les appels de procédure.

Les instructions de branchement, c'est-à-dire les sauts conditionnels ou non, transfèrent l'exécution du code vers un point arbitraire de la mémoire. Les sauts ont l'avantage d'un faible coût d'exécution mais posent un inconvénient majeur. Après un saut, il est impossible de connaître l'origine de celui-ci. Or, un aspect peut faire intervenir un grand nombre de points de jonction. Lorsque l'exécution d'un aspect est terminée, celle-ci doit se poursuivre après le point de jonction ayant déclenché l'aspect. Comme il n'est pas possible de connaître l'origine d'un saut, il faudrait dès lors, dupliquer le code des aspects pour chaque point de jonction. Une telle solution serait inadaptée à notre contexte d'étude.

Les interruptions sont un mécanisme de communication entre les programmes et le système d'exploitation. Elles peuvent être déclenchées soit par le matériel soit explicitement par le logiciel, c'est-à-dire via des instructions comme *int3*, ou implicitement lorsque des conditions particulières interviennent, par exemple une instruction pro-

cesseur invalide ou une violation de privilège. Le déclenchement d'une interruption est intercepté par le système d'exploitation, qui lorsque cela est nécessaire, déclenche l'exécution d'une routine particulière de l'application concernée. Par exemple, l'instruction processeur *int3* interrompt l'exécution et déclenche la routine du noyau associée au signal *sigtrap*, qui elle même déclenche la routine définie par le programme pour le signal *sigtrap*. L'instruction *int3* présente un intérêt pour le détournement du flot d'exécution : son opcode est codé sur un octet et peut donc se substituer à n'importe quelle instruction processeur. Contrairement aux branchements, les interruptions mémorisent la dernière instruction exécutée lors de leur déclenchement. Mais, le mécanisme d'interruption présente un défaut majeur pour le tissage de code en espace utilisateur. Les changements de contexte provoqués par l'instruction *int3* : application vers noyau puis noyau vers application, ont un coût important à l'exécution (plus de 10000 cycles processeur, cf section 6.2, tableau 6.1).

Les intructions d'appel de procédure, les *calls*, résolvent le problème des instructions de branchement : les *calls* mémorisent l'origine de l'appel (plus précisément l'adresse à laquelle retourner), tout en évitant le surcoût des changements de contexte provoqués par les interruptions. Néanmoins, les points de jonction pouvant se situer n'importe où dans l'espace mémoire de l'application et comme nous ne pouvons présupposer de l'emplacement du code des aspects, il est nécessaire d'utiliser un *call* à adressage direct sur 32 bits. Ce type d'appel est codé sur 5 octets, un pour l'opcode du *call* et quatre pour l'adresse de destination. Or, la plus petite instruction processeur est codée sur 1 octet. Néanmoins, nous verrons dans la section traitant de la localisation des points de jonction, que le plus petit des points de jonction correspondra toujours à une instruction codée sur plus de 5 octets. C'est donc le mécanisme des appels de procédure que nous avons choisi pour détourner le flot d'exécution aux points de jonction. Comme nous avons voulu l'architecture d'Arachne modulaire et extensible, le système propose également une librairie de réécriture utilisant l'interruption *int3*. Cette technique est uniquement utilisée dans l'extension d'Arachne (précédemment mentionnée dans la section 5.1.4) permettant l'interception des accès aux variables locales.

Nous allons maintenant voir comment nous garantissons la cohérence de l'exécution tout en remplaçant n'importe quelle instruction codée sur plus de quatre octets par un *call* à adressage direct sur 32 bits.

5.3.1.1 Réécriture sûre d'un point de jonction

Comme nous le verrons dans ce chapitre, les points de jonction du langage d'Arachne correspondent toujours à une unique instruction processeur de taille toujours supérieure ou égale à cinq octets. La réécriture d'un point de jonction consiste à transformer celui-ci en un appel à une procédure via un *call* à adressage direct.

La méthode de réécriture doit garantir la cohérence de l'exécution, c'est-à-dire que la réécriture ne doit pas perturber l'exécution du programme de base. De plus, la réécriture ne doit pas interrompre l'exécution du programme de base pour des raisons de performances.

Ce problème se réduit donc à garantir l'atomicité de la réécriture du point de jonction vis-à-vis du processeur. Les spécifications de l'architecture Intel 32 bits indiquent que la réécriture de quatre octets de code est atomique : si l'écriture de 4 octets mo-

difie une ou plusieurs instructions machine en cours d'exécution, l'exécution de ces instructions est invalidée et reprise après la dernière instruction exécutée.

Pour réécrire une instruction de cinq octets en un *call* à adressage direct, Arachne remplace les deux premiers octets de l'instruction par un saut court dont la destination est lui-même. Cette réécriture invalide potentiellement une exécution en cours du point de jonction et agit comme un verrou tournant. Arachne remplace alors les trois derniers octets du point de jonction par les trois octets de poids faibles de l'adresse de destination du futur *call*. Enfin, Arachne modifie à nouveau les deux premiers octets du point de jonction pour les remplacer par l'octet *e8* qui correspond à l'opcode du *call* et par l'octet de point fort de l'adresse de destination du *call*. Cette dernière écriture invalide l'exécution du verrou tournant et finalise la réécriture du point de jonction.

La réécriture d'instructions de plus de cinq octets fonctionne à l'identique. Par contre, le cas des aspects interceptant des appels de fonctions est légèrement différent. Dans ce cas les points de jonction étant des instructions *call*, il suffit de réécrire les quatre octets de l'adresse de la fonction appelée.

5.3.2 Les crochets – cohérence du déclenchement des aspects

Nous venons de voir comment garantir l'atomicité de la réécriture d'un point de jonction. Cela n'est pas suffisant pour garantir la cohérence du tissage. Les points de jonction une fois réécrits deviennent des appels de procédure. Nous verrons par la suite que la compilation des aspects encapsule le code des aspects dans une fonction C. Or, avant l'appel d'une fonction, il est nécessaire de mettre en place une pile et de sauvegarder certains registres du processeur. Le code des aspects doit également accéder au contexte d'exécution des points de jonction pour réaliser les tests dynamiques de la coupe. De plus, lorsque les conditions dynamiques de la coupe ne sont pas vérifiées, l'aspect doit exécuter le code original du point de jonction. Enfin, il est nécessaire que le tissage soit atomique vis-à-vis de l'exécution, c'est-à-dire qu'un aspect ne peut être exécuté que lorsque tous les aspects sont tissés.

Pour résoudre ces problèmes, nous utilisons des crochets. Comme le montre la figure 5.2, un crochet est un morceau de code assembleur qui est chargé au moment du tissage par le noyau de réécriture. Le noyau de réécriture crée un crochet pour chaque aspect. Ce crochet est appelé par les points de jonction et agit comme couche intermédiaire entre les points de jonction et les aspects. Lorsqu'un point de jonction appelle le crochet, ce dernier commence par sauvegarder sur la pile la totalité de l'état du processeur, c'est-à-dire les registres généraux et les registres à virgule flottante.

Ensuite, le crochet teste la garde de l'aspect. La garde est une variable utilisée comme verrou et est levée lorsque le tissage de l'aspect est terminé. Ainsi, lorsque la garde est bloquante, le crochet restaure à partir de la sauvegarde sur la pile, l'état du processeur, exécute le code original du point de jonction, et renvoie l'exécution vers l'instruction suivant le point de jonction. Dans le cas contraire, le crochet restaure l'état du processeur, et prépare le contexte d'exécution qui est passé au code de l'aspect. Lorsque l'exécution de l'aspect est terminée, le code retourne à l'instruction suivant le point de jonction. Ainsi, les crochets permettent de préparer l'exécution des aspects et de garantir l'atomicité du tissage.

Le cas des crochets pour des aspects interceptant des appels de fonctions présente une particularité intéressante. En effet, le point de jonction étant à l'origine un appel

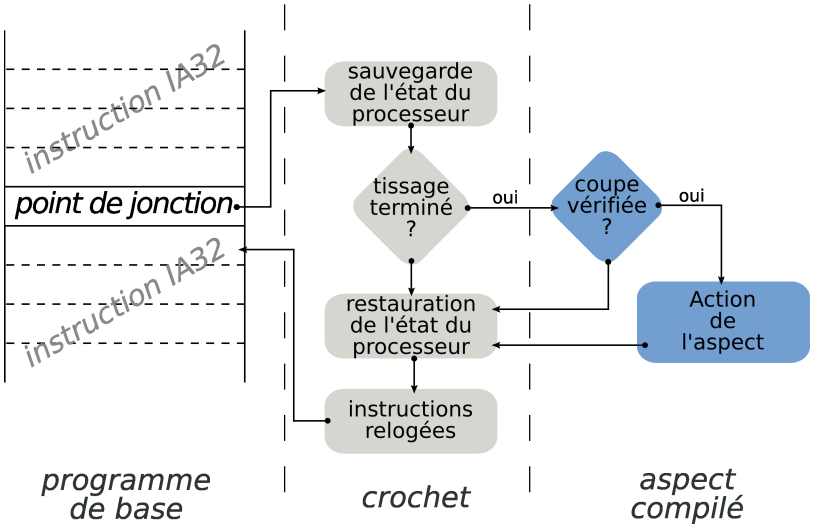


FIG. 5.2: Fonctionnement des crochets

```
.text
hook_template_start:
    pushl %eax
    pushf
0:
    // mov guard, %eax
    .byte 0xa1; .byte 0xff; .byte 0xff; .byte 0xff; .byte 0xff;
    cmpl $0xffffffff, %eax
    jne 2f
    popf
    popl %eax
    // jmp aspect
    .byte 0xe9; .byte 0xff; .byte 0xff; .byte 0xff; .byte 0xff;
2:
    popf
    popl %eax

    // jmp original call
    .byte 0xe9; .byte 0xff; .byte 0xff; .byte 0xff; .byte 0xff;
```

Listing 5.1: Un patron de crochet pour des aspects sur appels de fonction

de fonction et le code des aspects étant encapsulé dans une fonction dont le prototype est identique à celui de la fonction interceptée, le crochet n'a pas nécessité de préparer le contexte pour l'exécution de l'aspect. Le code du crochet est alors beaucoup plus simple. Le listing 5.1 montre un patron de crochet pour une coupe sur appels de fonction. On y trouve en entrée (lignes 3 et 4) la sauvegarde de l'état du processeur. Les lignes 6 à 9 correspondent au test de la garde qui déclenche soit l'aspect (ligne 13) soit l'appel original (ligne 19). Ici, les adresses de la garde, de l'aspect, et de la fonction interceptée ne sont pas renseignées, c'est le noyau de réécriture qui complète le crochet lors de sa création pendant la phase de tissage.

5.4 Compilation des aspects

Dans le chapitre 4, nous avons présenté le langage d'aspect défini pour le système Arachne. Ce langage comprend le langage C et est complété par un langage d'aspect associant des coupes et des actions. C'est-à-dire qu'un fichier d'aspect contient des

macros CPP (C Pre Processor), des définitions C (types, variables, fonctions) et des aspects dont les coupes et les actions peuvent référencer les définitions C contenues dans le fichier d'aspects.

La compilation d'un fichier d'aspects transforme un fichier contenant à la fois des macros, du code C et des aspects en un fichier d'aspects compilés contenant le code exécutable des aspects et les directives de tissage à fournir au noyau de réécriture. Comme le montre la figure 5.1, la compilation d'un fichier d'aspects est réalisée en plusieurs étapes. En premier lieu, *aac* utilise le pré-processeur *cpp* pour répandre les macros. Ensuite, le mélange de code C et d'aspects résultants est analysé par le parseur d'aspects. Celui-ci transforme les définitions des aspects en code C et en directives de tissage (sous la forme de structures de données C) normalisées pour le noyau de tissage. Le produit de cette transformation est un code source C. Ce code est ensuite compilé par *gcc* en une librairie partagée dynamique.

L'outil *aac* a été implémenté à partir de la librairie CAML FrontC [Cas], cette librairie est un analyseur lexical et syntaxique pour le C. Nous avons étendu cette librairie pour analyser le code source des aspects. L'analyse des aspects entraîne la génération de deux catégories de code ; Le code exécutable des aspects, c'est-à-dire le code qui sera déclenché lors de l'interception d'un point de jonction ; Et les directives de tissage contenant les informations nécessaires au noyau de réécriture pour relier le code exécutable des aspects avec le code du programme de base. Nous allons maintenant détailler ce travail.

5.4.1 Le code exécutable des aspects

Comme nous l'avons présenté dans le chapitre 4, notre langage d'aspect est construit au-dessus d'un modèle de points de jonction. Dans notre langage, un aspect associe une coupe avec une action. L'action peut être soit une expression C, soit un bloc d'expressions C. Le développeur de l'aspect peut référencer dans l'action, les données de la coupe, par exemple, pour une coupe sur un appel de fonction, l'action peut utiliser les paramètres de l'appel et également la fonction appelée (au travers du mot clé *proceed*). De plus, la coupe d'un aspect peut contenir des tests dynamiques complémentaires à l'interception des points de jonction. Il est donc nécessaire de générer un code exécutable par le programme de base qui réalise l'action de l'aspect ainsi que les tests dynamiques de la coupe.

Nous avons montré comment nous dévions le flot d'exécution au niveau d'un point de jonction par un appel de procédure vers un crochet, lequel permettait de réaliser les traitement nécessaires à l'isolation entre l'aspect et le programme de base afin de garantir la cohérence de l'exécution et à la mise en place d'un appel vers une fonction à laquelle passer les informations relatives au contexte d'exécution du point de jonction. Ces informations de contexte dépendent du type de point de jonction intercepté.

aac transforme chaque aspect en deux fonctions C. Les prototypes de ces deux fonctions dépendent de la coupe. La première de ces deux fonctions, appelée *pointcut*, est chargée de réaliser les tests dynamiques décrits dans la coupe de l'aspect. La seconde, appelée *advice*, encapsule l'action de l'aspect.

```
1 aspect exemple :: call (int foo (int a)) && if (a > 1)
  then {
      return proceed (--a);
  }
```

Listing 5.2: L'aspect « exemple » intercepte les appels à la fonction *foo*

```
1 static struct aspect_fcall exemple;

  int advice_exemple (int a) {
      int (*proceed)(int ) __attribute__ ((unused)) =
          (int(*)(int)) exemple.to_replace_ptr;
6
      return proceed(--a);
  }

  int pointcut_exemple (int a) {
11    int (*proceed)(int ) __attribute__ ((unused)) =
          (int(*)(int))exemple.to_replace_ptr;

      if(a > 1)
          return advice_exemple(a);
16    else
          return proceed(a);
  }
```

Listing 5.3: Code généré pour l'aspect « exemple » du listing 5.2

5.4.1.1 Aspects sur les appels de fonction

Considérons l'aspect du listing 5.2 nommé *exemple*, cet aspect intercepte les appels à la fonction *foo* lorsque son paramètre *a* est strictement supérieur à 1 (ligne 1). Lorsque la coupe de l'aspect *exemple* est vérifiée, l'aspect exécute l'appel à *foo* en décrémentant sa valeur de 1 (ligne 3). Le code exécutable généré pour cet aspect est présenté dans le listing 5.3. Ce code déclare un pointeur vers les directives de tissage propres à l'aspect *exemple* (ligne 1). Ensuite, on trouve deux fonctions dont les prototypes sont identiques à celui de la fonction *foo*. La première de ces fonctions, *advice_exemple* (ligne 3), correspond à l'action de l'aspect exemple. On retrouve dans son corps (ligne 7) le code de l'aspect *exemple*. On y trouve à l'identique l'appel à la fonction *foo* à travers l'identifiant réservé *proceed* qui est déclaré via une référence aux directives de tissage (ligne 4) qui sera résolue par le noyau de réécriture. On trouve ensuite la méthode *pointcut_exemple* (ligne 10) qui contient le test dynamique de la coupe de l'aspect *exemple*, ici la comparaison de l'argument *a* à 1 (ligne 14). Le résultat de la comparaison conditionne la suite de l'exécution : lorsque la coupe est vérifiée, on appelle l'action de l'aspect (ligne 15), dans le cas contraire, on exécute le point de jonction original (ligne 17).

5.4.1.2 Aspects sur les flots de contrôle

Considérons l'exemple du listing 5.4. Cet exemple est très similaire à celui du listing 5.2, à la différence cette fois que l'appel à la fonction *foo* fait partie d'un flot de contrôle comprenant les fonctions *zoo* et *bar* (ligne 1). Le code généré pour cet aspect est présenté dans le listing 5.5. Ce code est quasiment identique au cas vu précédemment. La seule différence réside dans le test dynamique de la coupe qui contient un test additionnel réalisé par la fonction *stack_check* (ligne 13).

Ce test vérifie que la pile d'appel du programme contient bien le flot de contrôle de l'aspect. C'est-à-dire les fonctions *zoo* et *bar*. Les spécifications du langage C imposent que les fonctions créent une nouvelle fenêtre (frame) dans la pile quand elles sont

```
aspect exemple :: controlflow (int zoo (), int bar (),
    call (int foo (int a)) && if (a > 1))
then {
    return proceed (--a);
}
```

Listing 5.4: L'aspect « exemple » intercepte les appels à la fonction *foo* lorsqu'elle est appelée dans *bar* et *zoo*

```
static struct aspect_cflow exemple;

int advice_exemple(int a) {
    int (*proceed)(int ) __attribute__ ((unused)) =
        (int(*)(int)) exemple.final.function_call.to_replace_ptr;
    return proceed(--a);
}

int pointcut_exemple(int a) {
    int (*proceed)(int ) __attribute__ ((unused)) =
        (int(*)(int)) exemple.final.function_call.to_replace_ptr;

    if(stack_check(exemple.function_stack, 1) && a > 1)
        return advice_exemple(a);
    else
```

Listing 5.5: Code généré pour l'aspect « exemple » du listing 5.4

appelées. La vérification du flot de contrôle est réalisée en inspectant le contenu de la pile.

5.4.2 Les directives de tissage

Nous avons voulu modulariser le système Arachne et pour rendre le processus de compilation des aspects ainsi que les aspects compilés indépendants des techniques de tissage et du programme de base. Pour cela, il était nécessaire que les aspects compilés embarquent les informations nécessaires au noyau de tissage. De plus, comme nous le montre le listing 5.3 (ligne 4), il est nécessaire d'abstraire les références au programme de base dans les aspects compilés. Pour cela, les aspects compilés embarquent des directives de tissage.

5.4.2.1 Aspects sur les appels de fonction

Les directives de tissage se présentent comme le montre le listing 5.6 (suite du listing 5.3) sous la forme de structures de données C. Dans tout fichier d'aspects compilés, on retrouvera toujours la variable *AspectInFile* (ligne 29). *AspectInFile* est un tableau de pointeurs sur des directives de tissage. Cette variable est le point d'entrée utilisée par le noyau de réécriture pour parcourir l'ensemble des directives de tissage du fichier à tisser. Dans notre exemple, le fichier contient un unique aspect : *exemple*, pour lequel les directives de tissage sont référencées par la structure de données de type *asepct_fcall* portant le même nom que l'aspect (ligne 20).

Les trois premiers champs des directives de tissage sont communs à tout type d'aspects. On trouve le type de l'aspect (ici *FCALL*). Ce champs indique au noyau de réécriture comment interpréter le reste de la directive. Ensuite, on trouve le nom de l'aspect et l'adresse du crochet d'interception des points de jonction. Ce dernier champs sera renseigné à la volée par le noyau de tissage.

On trouve ensuite les informations spécifiques aux aspects interceptant les appels de fonctions. En premier, le nom de la fonction dont les appels sont interceptés (ici *foo*,

```
static struct aspect_fcall exemple = {
    FCALL,
    "exemple",
    (void*) 0,
    "foo",
    (void*) 0,
    pointcut_exemple,
};

aspect_t* AspectInFile[] = {
    (aspect_t*) &exemple,
    (aspect_t*) 0
};
```

Listing 5.6: Directives de tissage pour l'aspect « exemple » du listing 5.2

```
static struct aspect_cflow exemple = {
    CFLOW,
    "exemple",
    (void*) 0,
    FCALL,
    {
        {FCALL,
        "exemple",
        (void*) 0,
        "foo",
        (void*) 0,
        pointcut_exemple},
    },
    1,
    (void*) 0,
    {"bar", "zoo", 0,},
};

aspect_t* AspectInFile[] = {
    (aspect_t*) &exemple,
    (aspect_t*) 0
};
```

Listing 5.7: Directives de tissage pour le flot de contrôle « exemple » du listing 5.4

ligne 24). Puis, l'adresse de cette fonction, ici initialisée à zéro et qui sera renseignée au moment du chargement de l'aspect par le noyau de réécriture. Finalement, on trouve l'adresse du point d'entrée du code exécutable de l'aspect. Cette adresse est utilisée pour lier l'aspect au crochet.

5.4.2.2 Aspects sur les flots de contrôle

Le listing 5.7 présente les directives de tissage générées pour l'aspect du listing 5.4. Les directives pour un aspect sur flot de contrôle reprennent le même en-tête. On trouve d'abord le type de l'aspect ici *CFLOW*, ensuite le nom de l'aspect et l'adresse du crochet. Ensuite on trouve les informations relatives à la dernière étape du flot de contrôle, ici l'appel à *foo* (lignes 23 à 31). Enfin, on trouve les informations spécifiques au flot de contrôle ; La valeur 1 indique que c'est un flot de contrôle strict (ligne 32) ; Le contenu du flot de contrôle (ligne 34) ; Et les adresses des fonctions du flot de contrôle qui seront résolues lors du tissage (ligne 33).

5.5 Résolution des informations de tissage

L'un des points forts de l'architecture du système Arachne est de rendre les aspects compilés indépendants du programme à tisser. Néanmoins, les aspects étant des morceaux de code venant interagir avec l'exécution du programme de base, le code

des aspects utilise des références explicites au programme à tisser. Par exemple, un aspect sur appel de fonction fait référence au programme de base à travers le nom d'une fonction à intercepter, de même la construction *require* du langage d'Arachne permet, comme le montre le listing 4.2, de réutiliser une variable ou une fonction du programme à tisser dans le code des actions des aspects. De même, les coupes des aspects sélectionnent des points de jonction auxquels le tisseur d'aspect doit injecter les aspects.

Dans les deux cas (réutilisation du programme de base et points de jonction), le tisseur d'aspect doit après le chargement des aspects, résoudre les références au programme de base faites par les aspects. Nous allons maintenant présenter comment fonctionne cette résolution, d'abord dans le cas des références au programme de base, puis dans le cas de la localisation des points de jonction.

5.5.1 Résolution des symboles

La résolution des symboles du programme de base, données ou fonctions, requiert de convertir le nom du symbole par une adresse mémoire à laquelle se situe le symbole. Les techniques utilisées pour résoudre les sysmboles du code à tisser varient selon que le code est un programme utilisateur ou le noyau Linux.

En espace utilisateur Dans le cas des programmes en espace utilisateur, la mise en relation des symboles du code source avec des adresses mémoire est une condition primordiale pour le fonctionnement des débogueurs, l'édition des liens ou encore l'interopérabilité entre les compilateurs. De ce fait, nous avions vu dans le chapitre 4 que dans l'environnement Linux, la norme ELF [Sta95] imposait un format de représentation de tout programme à la fois sur disque et lors de l'exécution. Afin de respecter ces normes, les programmes s'exécutant sous Linux utilisent majoritairement le format ELF, qui nécessite de la part du compilateur d'embarquer dans le programme compilé, des tables associant les noms des symboles du programme avec leur adresse mémoire lors de l'exécution. En pratique, ces tables référencent tous les symboles du programme déclarés *publiques*, c'est-à-dire toutes les fonctions et toutes les variables n'ayant pas été déclarées explicitement avec le mot clé *static*. Ces informations étant standardisées, nous utilisons la librairie BFD, *Binary File Descriptor Library*, fournie dans la collection d'outils *Binutils* [Foub], pour résoudre les adresses des symboles référencés dans les aspects. Cette approche suppose que l'exécutable du programme de base soit accessible sur le disque. De plus, cette technique est incompatible avec des programmes *stripés*, c'est-à-dire dont les tables des symboles ont étés supprimées pour réduire la taille de l'exécutable.

En espace noyau Bien que le noyau Linux soit un exécutable compilé au format ELF comme le plus souvent pour les applications Linux, il n'est pas possible d'utiliser directement la technique précédente pour le noyau. Afin de réduire le temps de *boot* du noyau mais également pour respecter des limites de taille imposées par les *boot loader*, le noyau Linux est *stripé* et compressé sur le disque. Il est néanmoins possible lors de la compilation d'un noyau de récupérer une copie de celui-ci avant qu'il soit *stripé* et compressé, et d'utiliser la librairie BFD comme précédemment. Néanmoins,

à l'exception de la distribution *Gentoo*, la majorité des distributions Linux fournissent des noyaux pré-compilés.

Pour qu'il puisse être débogué et que l'on puisse lui adjoindre dynamiquement des modules, le noyau Linux fournit deux méthodes pour la résolution des symboles.

La première est fournie au travers du fichier *System.map* communément situé à la racine du système ou dans le répertoire *boot*. Ce fichier est créé lors de la compilation du noyau avant la suppression des informations d'édition des liens (*stripping*). Il correspond à la sortie du programme *nm* qui liste l'ensemble des symboles d'un exécutable, sur l'exécutable noyau. Ce fichier présente une limitation : il ne liste que les symboles du noyau et pas ceux contenus dans les modules.

La deuxième méthode utilise le fichier *kallsyms* du pseudo système de fichier *procfs*. Ce fichier contient les symboles présents dans le *System.map* mais également les symboles contenus dans les modules chargés dynamiquement. Notre choix s'est donc porté sur ce fichier pour la résolution des symboles du noyau Linux.

5.5.2 Localisation des points de jonction

Nous avons pu voir précédemment que la compilation des aspects transformait chaque aspect en deux fonctions C. La première contient le code de l'action de l'aspect et est appelée par la seconde. Cette dernière contient les tests dynamiques permettant de vérifier le déclenchement des aspects et est appelée par le crochet d'interception du flot d'exécution qui est lui même exécuté au niveau de tous les points de jonction sélectionnés par la coupe.

Lorsqu'un aspect compilé est chargé dans l'espace mémoire du programme de base, il est d'abord lié à un nouveau crochet. Puis, le noyau de tissage dévie le flot d'exécution au niveau de tous les points de jonction. Dans ce but, le noyau de réécriture doit localiser tous ces points.

Les points de jonction dépendent du type d'aspect tissé, par exemple les appels à une fonction ou les lectures de la valeur à une adresse mémoire. Comme nous l'avons expliqué dans le chapitre 4, les points de jonction utilisés dans le langage d'aspect d'Arachne sont soit des appels de fonction, soit des accès à des variables globales. Comme nous l'avons vu précédemment, le code binaire généré pour les appels de fonction ou pour les accès à des variables globales, est standardisé par l'*Application Binary Interface* et le format ELF.

Les appels de fonctions Les appels de fonctions C sont impérativement compilés sous la forme de l'opcode *call* dans son adressage direct, qui correspondent aux *bytecode* `e8` suivi de l'adresse sur 32 bits de la fonction interceptée. Pour rappel, cette règle ne s'applique pas aux fonctions statiques ou inlinées ni aux appels de fonctions à travers un pointeur de fonction.

Les accès aux variables globales Contrairement aux appels de fonction, il existe une multitude d'instructions processeurs réalisant des accès à des variables globales. De même, bien que les variables globales C doivent être impérativement stockées en mémoire, il n'est pas rare qu'une variable soit transférée dans un registre afin d'effectuer un calcul intermédiaire. Néanmoins, l'ABI IA32, impose deux restrictions sur

l'utilisation des variables globales. D'une part, à la fin de tout *statement* C (c'est-à-dire pour tout point-virgule des sources) le contenu de toutes les variables globales du programme doit être à jour. D'autre part, l'accès à une variable globale en mémoire au niveau du source, doit toujours se traduire par un accès par adressage direct au niveau du code binaire. De ce fait, les accès aux variables globales correspondent à tout opcode ayant un paramètre référencé par un adressage direct.

Implémentation de la résolution des points de jonction Nous venons de voir que la localisation des points de jonction nécessitait l'analyse du code binaire généré par le compilateur C. Cette analyse nécessite de parcourir le code binaire à la recherche d'opérations spécifiques dans des adressages particuliers. Pour réaliser cette tâche nous avons utilisé la librairie *libasm* conçue pour l'outil ELFsh faisant partie du projet ERESI, *Reverse Engineering Software Interface* [teab]. Cette librairie permet le désassemblage de code binaire pour les processeurs IA32 et présente pour chaque instruction processeur ses informations détaillées, par exemple le type d'adressage utilisé pour les arguments.

5.6 Conclusion

Dans ce chapitre, nous avons couvert l'essentiel de l'implémentation du système de tissage Arachne. Notre mise en œuvre s'appuie sur les standards du langage C, de la plateforme Intel 32 bits et du système d'exploitation Linux. L'utilisation des crochets permet un tissage atomique sans interruption de l'exécution du programme de base. De plus, comme nous le démontrons dans le chapitre suivant, les informations de contexte et la connaissance de la sémantique du programme et du langage C, nous permettent de détourner le flot d'exécution de manière performante en comparaison des techniques génériques (principalement l'utilisation de l'interruption 3) utilisées dans les outils similaires.

There are no bugs, only
unrecognized features.

Thoreau's Theories of Adaptation

Chapitre 6

Expérimentations et évaluations

Où nous quantifions avec précision les coûts de déclenchement de tous les types d'aspect [DFL+06]. Pour cela, nous y définissons un protocole expérimental pour la mesure de courtes durées d'exécution. Nous y démontrons également l'applicabilité en termes d'expressivité et de performance, de la programmation par aspects pour un large panel d'évolutions sur des applications réelles : intégration d'une politique de préchargement dans le cache Web Squid [SDML+06], correction de bogues dans wu-ftpd [LSDM05a], monitoring conjoint de Squid et du noyau Linux [LM07], etc.

Sommaire

6.1 Protocole expérimental **76**
6.1.1 Difficultés 76
6.1.2 Protocole 77
6.1.3 Validation 78
6.2 Micro-évaluations **78**
6.3 Macro-évaluations **81**
6.3.1 Ajout d'une politique de préchargement dans Squid 81
6.3.2 Corrections de trous de sécurité 83
6.3.3 Supervision des accès aux systèmes de fichiers 86
6.4 Conclusion **87**

Nous venons de présenter l'implémentation de notre système de tissage d'aspects à la volée, Arachne. Notre mise en œuvre utilise des techniques d'injection dynamique de code. Nous avons pu voir que l'injection de code est rendue difficile par les compilateurs, qui réalisent de nombreuses optimisations du code binaire. Afin de contourner les difficultés créées par les optimisations, nous avons utilisé des techniques de *reverse-engineering*. Ces contournements opposent le code injecté à la volée et les optimisations réalisées par le compilateur. Or, le choix du langage C en programmation système reflète la volonté des développeurs de rester proche de la machine et de maîtriser finement le code machine généré. Ainsi, il est nécessaire pour démontrer la praticabilité d'Arachne pour la maintenance et l'évolution de logiciels système, de prouver que l'impact de l'injection de code sur les performances est limité et compatible avec les objectifs des développeurs système.

Dans cette partie, nous présentons les scénarios d'évaluation des performances de notre système de tissage d'aspects à la volée. Du fait de la complexité des architectures

matérielles, il est difficile de mesurer l'impact local (c'est-à-dire au niveau d'un point de jonction) d'Arachne sur les performances du code machine. En effet, la gestion du cache processeur, les temps d'exécution du code machine qui varient selon les contextes et les optimisations à la volée par le processeur du code machine complexifient l'évaluation des performances d'Arachne. Dans un premier temps, nous présentons les techniques que nous avons employées pour réaliser nos évaluations et la manière dont nous avons validé notre protocole expérimental. Ensuite, nous mesurons l'impact d'Arachne sur les performances locales du code machine. Enfin, nous montrons qu'Arachne satisfait les exigences de performance recherchées par les développeurs d'applications système à travers les scénarios d'utilisations réelles sur des applications en production, discutés dans le chapitre 3.

6.1 Protocole expérimental

Nous présentons ici le protocole expérimental que nous avons suivi lors de la réalisation de nos tests du système Arachne. En premier lieu, nous identifions les difficultés liées à la mesure des durées d'exécution des points de jonction. Puis, nous présentons les solutions que nous avons mises en œuvre pour contourner ces difficultés. Enfin, nous confrontons notre protocole expérimental aux spécifications des processeurs Intel 32 bits.

6.1.1 Difficultés

Afin de mesurer le coût d'Arachne sur les performances d'une application, nous allons étudier à la fois son impact sur l'exécution du code machine et sur les performances d'applications en production. Nous avons vu que pour injecter un aspect dans une application, Arachne dévie artificiellement le flot d'exécution de l'application en remplaçant les points de jonction par des instructions déclenchant les aspects. Il nous faut donc comparer le temps d'exécution d'un point de jonction avec sa version tissée. Or, de telles mesures de temps sont extrêmement difficiles à réaliser. Les points de jonction correspondent à des instructions machines très optimisées. Or, les API Linux de mesure de temps (*time*, *clock*, *etc*) offrent des résolutions d'horloge de l'ordre de la milliseconde. Les processeurs récents affichant allègrement des cadences de 3GHz, ces API sont clairement inadaptées à la mesure des temps d'exécution des points de jonction.

En plus de présenter des temps d'exécution courts, ces durées dépendent du contexte d'exécution du processeur. Deux points rentrent principalement en jeu : le cache d'instructions et le cache de données. En effet, la durée d'exécution des instructions machine de branchement, c'est-à-dire les sauts (*jmp*), les sauts conditionnels (*bne*, *beq*...), et les branchements sur les conditions d'erreur (lors de l'exécution d'une instruction invalide ou d'une violation de droits), varie selon que le code à la destination du branchement est chargé dans le cache d'instruction et dépend de l'évaluation de la condition de branchement. De même, le temps d'exécution des instructions à un ou plusieurs arguments non-constants dépend du contenu du cache de données.

Enfin, la définition de notre protocole expérimental se confronte aux optimisations du code machine réalisées à la volée par le processeur. D'une part, le processeur peut réarranger à la volée l'ordre d'exécution des instructions. D'autre part, le temps

d'exécution d'une instruction machine dépend des instructions l'entourant. En effet, les processeurs modernes utilisent un ou plusieurs pipelines d'instructions, c'est-à-dire que l'exécution à proprement parlé des instructions machine est décomposée en plusieurs étapes (chargement de l'instruction, décodage de l'instruction, chargement des paramètres, calculs, retour des résultats...). Lorsqu'elles sont indépendantes, le processeur peut exécuter plusieurs instructions simultanément. Dès lors, la durée d'exécution d'une instruction dépend des instructions qui la précédent. La « profondeur » du pipeline définit le nombre maximum d'instructions pouvant s'exécuter simultanément. Les processeurs Intel Pentium 4 Prescott possèdent des pipelines de profondeur trente-et-un [Int01]. Ce cas reste néanmoins exceptionnel et les constructeurs développent des pipelines d'une profondeur d'environ vingt, car il est difficile d'exploiter pleinement des pipelines de profondeur supérieure.

6.1.2 Protocole

Nous avons identifié quatre points de difficulté dans la mesure des temps d'exécution des points de jonction : les durées d'exécution trop courtes pour être mesurées par des API standard – le réordonnancement des instructions machine par le processeur – les variations des temps d'exécution des instructions – le contexte d'exécution (défauts de cache). Nous avons adapté notre protocole de mesure pour minimiser l'impact de ces éléments durs.

Pour mesurer des temps réduits, nous avons choisi d'utiliser l'instruction machine *rdtsc* (Read Time-Stamp Counter). Cette instruction retourne le contenu du registre de 64-bits *TimeStampCounter* dans les registres généraux *edx eax*, le registre *TimeStampCounter* compte les cycles processeurs écoulés depuis la mise en route de la machine. Le cycle est la plus petite unité de mesure du temps. C'est donc la plus adaptée à la mesure des durées d'exécution des instructions machine.

Pour empêcher le réordonnancement des instructions machine dont on veut mesurer l'exécution, nous utilisons l'instruction machine *mfence* (Memory Fence). Cette instruction bloque l'exécution du code machine tant que des opérations de lecture d'arguments et/ou d'écriture d'arguments effectuées par des instructions dans le pipeline ne sont pas terminées. L'instruction *mfence* force ainsi l'attente de la terminaison des instructions précédentes et empêche leur réordonnancement avec les instructions suivantes. Pour assurer que seules les instructions évaluées font partie de la mesure, nous forçons la terminaison des instructions avant le début et la fin de la mesure (listing 6.1).

Pour compenser les variations des temps d'exécution, nous répétons nos mesures autant de fois que nécessaire. Afin d'écarter les mesures erronées dues à la préemption du code de test pour une autre application, nous utilisons l'API *getrusage* pour écarter les mesures indésirables. Cette API nous renseigne sur le nombre de préemptions, de défauts de caches, *etc* survenus depuis le démarrage de l'application.

L'examen de l'exécution d'un programme entraîne la modification de son comportement. Notre protocole d'évaluation n'échappe pas à ce principe parfois appelé Heisenbug par référence au principe d'incertitude énoncé par le physicien Heisenberg [Lap90]. Notre protocole ne mesure pas uniquement la durée d'exécution du point de jonction mais également celle de l'instruction *mfence* forçant la sérialisation avant la fin de la mesure et celle de la sauvegarde de la valeur de retour de *rdtsc* dans la fonction *startTimer* (cette sauvegarde n'est pas explicitement écrite mais générée par le compilateur).

```
typedef unsigned long long int timer;

__attribute__ ((always_inline))
static void startTimer (timer* t) {
    __asm__ __volatile__ ("rdtsc\n\n"
    "mfence\n\t"
    : "=A" (*t));
}

__attribute__ ((always_inline))
static void stopTimer (timer* t) {
    __asm__ __volatile__ ("mfence\n\n"
    "rdtsc\n\t"
    : "=A" (*t));
}

void test () {
    timer start, end;

    startTimer (&start);
    /* Instructions */
    stopTimer (&end);
    fprintf(stdout, "%lli\n", end-start);
}
```

Listing 6.1: Intructions de mesure des cycles écoulés et de sérialisation

Afin de limiter l'impact de ces instructions de mesure dans la mesure elle-même, nous mesurons plusieurs instances des instructions à évaluer, pour que cet impact soit ventilé par les répétitions. Pour cela, on encadre les instructions évaluées dans une boucle dont on force le déroulement par le compilateur.

Enfin, dans le but de réduire la probabilité que les instructions mesurées soient perturbées par un défaut de cache, nous forçons l'alignement de ces instructions dans le code (macro *align* listing 6.2 ligne 24, 4096 étant la taille d'une page). Toutes les mesures présentées dans ce document ont été obtenues sur une machine équipée d'un Intel Pentium 4 cadencé à 3,3GHz et de 1Go de mémoire RAM. L'environnement d'exécution est un noyau Linux 2.6.

6.1.3 Validation

Afin d'évaluer la qualité de nos mesures, nous avons confronté notre protocole à la mesure d'un temps connu par avance. Les spécifications des processeurs Intel donnent les temps moyens d'exécution en cycles d'horloge pour chaque instruction. Dans un contexte d'exécutions spéculatives et de réordonnancement des instructions, ces temps ont une valeur toute relative. Néanmoins, celles-ci sont suffisantes pour vérifier la validité de notre protocole expérimental. Ainsi, nous avons utilisé notre protocole expérimental pour mesurer la durée d'exécution d'une instruction *nop* (*No Operation*). Cette instruction qui force le processeur à ne rien faire, est en fait un alias de l'instruction *xchg %eax, %eax* (échanger le contenu des registres *eax* et *eax*) et s'exécute en moyenne en 1 cycle d'horloge [Int01]. La mesure obtenue avec notre protocole expérimental est de 1 cycle d'horloge avec un écart type à la moyenne inférieur à 1,6%. Ce résultat confirme ainsi la validité de notre protocole expérimental.

6.2 Micro-évaluations

Dans cette évaluation, nous nous sommes concentrés sur l'étude du coût de chaque construction de notre langage d'aspect. Le langage d'aspect d'Arachne est construit sur

```
typedef unsigned long long int timer;

__attribute__ ((always_inline))
static void startTimer (timer* t) {
    __asm__ __volatile__ ("rdtsc\n\n"
    "mfence\n\t"
    : "=A" (*t));
}

__attribute__ ((always_inline))
static void stopTimer (timer* t) {
    __asm__ __volatile__ ("mfence\n\n"
    "rdtsc\n\t"
    : "=A" (*t));
}

void test () {
    int i, j;
    timer start, end;
    struct rusage pre, post;

    for (i = 0; i < OUTER_ITER; i++) {
        getrusage(RUSAGE_SELF, &pre);
        __asm__ __volatile__ (".align 4096\n");
        startTimer (&start);
        for (j = 0; j < INNER_ITER; j++) {
            /* Instructions */
        }
        stopTimer (&end);
        getrusage(RUSAGE_SELF, post);

        if (pre.ru_nivcsw == post.ru_nivcsw && pre.ru_majflt == post.ru_majflt)
            fprintf(stdout, "%lli\n", end-start);
        else
            fprintf(stderr, "preempted %li, pagefault %li, %lli cycles\n",
                post.ru_nivcsw-pre.ru_nivcsw,
                post.ru_majflt-pre.ru_majflt,
                end-start);
    }
}
```

Listing 6.2: Protocole expérimental pour la mesure de durées en cycles processeur

un langage de coupe qui permet de sélectionner des points de jonction sur lesquels tisser les aspects, c'est-à-dire, les instructions machine devant être interceptées pour exécuter les aspects. Notre but est ici de caractériser avec précision le coût du déclenchement d'un aspect. Dans le chapitre précédent, nous avons montré que l'injection des aspects était réalisée par réécriture du code machine au niveau des points de jonction.

Pour évaluer le coût des constructions de notre langage, nous avons écrit un aspect pour chaque construction du langage (appel de fonction, accès à une variable, *etc*) dont l'action est vide. Ces aspects se comportent comme des interpréteurs du programme de base. En effet, quand une action est vide, après interception du point de jonction, l'aspect exécute le code original. Par exemple, pour évaluer la construction *call*, nous avons écrit un aspect interceptant l'appel à une fonction et dont l'action consiste à appeler cette fonction.

Pour chaque construction du langage, nous avons comparé les temps d'exécution du point de jonction et du déclenchement de l'aspect. Par exemple, dans le cas des *call*, nous avons mesuré la durée d'exécution d'un appel à une fonction vide[1] dans sa version native et dans sa version tissée. Le tableau 6.1 présente les temps d'exécution mesurés et les ratios entre les exécutions natives et tissées.

La mesure réalisée pour la construction *seq* correspond à l'interception des appels

[1]Vide non pas au sens du langage C, mais au sens assembleur, c'est-à-dire que la fonction ne contient que l'instruction machine *ret*.

	Execution times (cycles)		
	Arachne	Native	Ratio
`call`	$28^{\pm 2.3\%}$	$21^{\pm 1.9\%}$	1.3
`seq`	$201^{\pm 0.5\%}$	$63^{\pm 1.7\%}$	3.2
`cflow`	$228^{\pm 1.6\%}$	$42^{\pm 1.8\%}$	5.4
`readGlobal`	$2762^{\pm 4.3\%}$	$1^{\pm 0.2\%}$	2762
`read`	$9729^{\pm 4.9\%}$	$1^{\pm 0.6\%}$	9729

TAB. 6.1: Mesures et Ratio entre exécution native et interceptée pour chaque construction du langage de coupe

successifs à 3 fonctions. Tandis que le *control flow* mesuré intercepte l'appel à une fonction depuis le corps d'une autre. Enfin, pour les constructions *readGlobal* et *read*, nous avons mesuré la lecture d'une variable globale lorsque celle-ci est affectée à une variable locale. Le compilateur C produit pour ce type d'accès, une instruction machine *mov*. Parmi les nombreuses instructions machine accédant à une variable en lecture, celle-ci présente la plus courte durée d'exécution (1 cycle lorsque la taille de la donnée lue n'excède pas la taille des mots du processeur : 32 bits dans notre cas), c'est le cas le plus défavorable pour Arachne.

On constate que les ratios varient de 1,3 dans le cas d'une construction *call* à un peu moins de 10000 pour un *read*. Étant donné que l'action d'un aspect est compilée sous la forme d'une fonction C et qu'il n'est pas nécessaire de réaliser des tests dynamiques pour valider l'interception d'un point de jonction de type *call*, il était relativement prévisible que le surcoût à l'exécution d'une coupe sur un appel de fonction soit négligeable. De même, les coûts induits par les constructions *seq* et *control flow* restent limités. En effet, l'interception de ces points de jonction ne diffère du cas des *call* que dans la vérification dynamique de la coupe. Par exemple, dans le cas de la séquence : l'avancement d'une étape dans le suivi de séquence (qui consiste en un échange de pointeurs entre deux listes chaînées) ou, dans le cas du *control flow* : la vérification du contenu de la fenêtre de pile précédente.

De même, il était prévisible que l'interception d'un accès à une variable exhibe un surcoût important (2762). L'interception de telles instructions nécessite de sauvegarder de nombreuses données en prévision de l'exécution de l'aspect (registres généraux, pointeur d'instructions, *etc*). De même, le mécanisme des signaux utilisé dans le cas de *read* implique une interruption pour violation de protection interceptée par le système d'exploitation qui la fait suivre par un signal à Arachne. Cet aller-retour entre le code utilisateur et le code du noyau explique largement le surcoût important de cette construction (9729).

Comme nous l'avons vu dans le chapitre 4, les constructions de type *seq* et *control flow* peuvent référencer plusieurs coupes (plusieurs étapes dans le cas d'une séquence et plusieurs appels imbriqués dans le cas d'un flot de contrôle). Les temps d'exécution de telles coupes dépendent linéairement du nombre de séquences en cours d'interception et respectivement du nombre d'appels imbriqués à vérifier (mise à jour des données de la séquence et respectivement vérification des *frame* de pile précédentes). De même, l'exécution d'un accès à une variable dépend de la taille de celle-ci. En effet, sur des processeur IA32, l'accès à une donnée de taille inférieure à 32 bits est exécuté en 1 cycle, alors que pour des tailles supérieures le temps d'accès augmente linéairement.

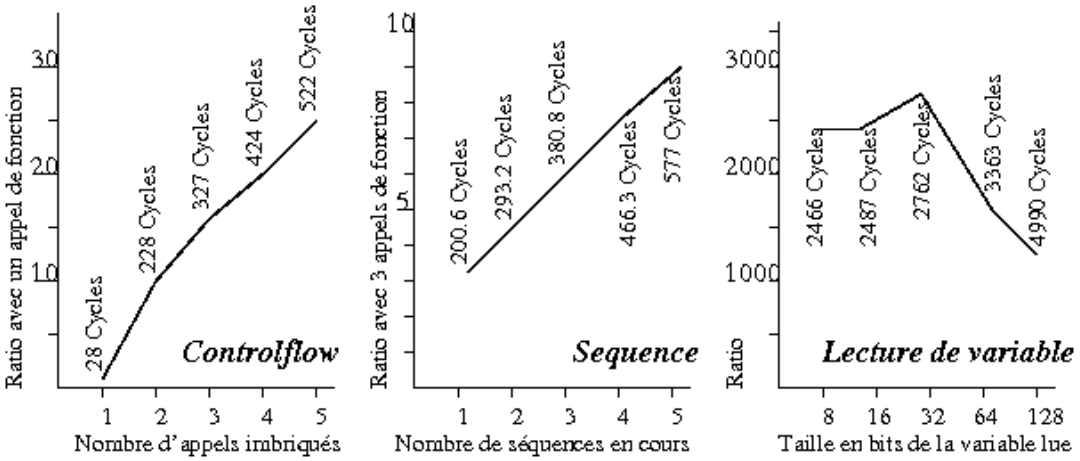


FIG. 6.1: Performance des contructions *seq*, *control flow* et *readGlobal*

La figure 6.2 confirme ces tendances.

6.3 Macro-évaluations

Nous venons de présenter l'impact local de notre tisseur d'aspect sur les temps d'exécution du code machine. Ces résultats permettent à un développeur d'aspect de quantifier finement l'impact d'un aspect sur l'exécution d'un point de jonction. Ces coûts variant de quelques cycles dans le cas d'un point de jonction sur un appel de fonction à quelques milliers de cycles pour un point de jonction sur l'accès à une variable, l'impact d'Arachne sur les applications dépend des aspects tissés. Aussi, nous allons évaluer le coût induit par Arachne lors de l'injection d'évolutions concrètes sur des applications en production.

Dans les chapitres 3 et 4, nous avons présenté six exemples (correction de l'allocateur de mémoire C, ajout d'une politique de préchargement dans Squid, remplacement du protocole TCP par UDP, correction de débordement de tableaux, détection de l'interblocage de verrous et supervision de Squid) illustrant le langage conçu pour Arachne. Ces exemples couvrent les différents domaines de l'évolution logicielle que sont l'ajout de fonctionnalité, la correction d'erreurs et l'adaptation à l'environnement.

Pour évaluer Arachne dans le cadre d'utilisations concrètes, nous mettons en œuvre certains des exemples déjà introduits. Nous présentons les résultats obtenus dans le cadre de l'ajout d'une politique de préchargement dans le cache Web Squid, de la correction de trous de sécurité critiques dans des applications libres et de la supervision des accès aux systèmes de fichiers par un cache Web dans le noyau Linux.

6.3.1 Ajout d'une politique de préchargement dans Squid

Les caches Web sont des intermédiaires réseau qui enregistrent le contenu Web téléchargé par les utilisateurs d'un réseau. En ayant connaissance de la sémantique du contenu Web téléchargé, un cache Web peut encore améliorer la latence perçue par les utilisateurs en préchargeant du contenu en fonction des requêtes précédentes. Par

Taille (ko)	Client (sec)	Cache (sec)	Prefetching (μsec)	Arachne (μsec)
3.8	<0.1	<0.1	2	0.008
46	<0.1	<0.1	39	0.01
70	0.1	<0.1	113	0.014
90	<0.1	<0.1	122	0.016
163	<0.1	<0.1	154	0.028
196	0.2	0.2	365	0.055
301	0.1	<0.1	43	0.044
1182	0.6	0.6	924	0.141
3875	1.3	1.3	16679	0.677

(a) Pages non préchargées

Client (sec)	Cache (sec)	Prefetching (μsec)	Arachne (μsec)
<0.1	<0.1	2	0.003
<0.1	<0.1	39	0.007
<0.1	<0.1	110	0.011
<0.1	<0.1	122	0.014
<0.1	<0.1	283	0.045
0.2	0.2	356	0.020
<0.1	<0.1	42	0.051
0.5	0.5	919	0.142
1	1	1801	0.446

(b) Pages en cache

TAB. 6.2: Répartition du temps d'exécution entre Squid, Arachne et la politique de préchargement. Toutes les valeurs sont des moyennes sur 200 mesures. La colonne *taille* indique le poids de la page téléchargée. La colonne *client* montre le temps observé par le client pour la récupération de la page. La colonne *Cache* présente le temps mesuré par Squid de traitement de la requête. La colonne *Prefetching* indique le temps d'exécution passé dans la politique de préchargement. Enfin, la colonne *Arachne* montre le temps d'exécution passé dans le code spécifique aux aspects : crochets et tests dynamiques.

exemple, lors du téléchargement d'une page Web, le cache peut précharger les pages et les images référencées par celle-ci.

Dans la section 4.4.2, nous avons présenté l'implémentation de la politique de préchargement de Ken-Ichi Chinen et Suguru Yamaguchi [CY97] dans Squid à l'aide du langage d'aspect d'Arachne. Nous réutilisons ici cet exemple pour évaluer le coût des aspects sur l'exécution d'une application tissée. Nous utilisons ici un Squid augmenté à la volée de la dite politique. Nous avons mesuré le temps d'exécution passé dans le code de gestion des aspects (c'est-à-dire l'interception des points de jonctions, les tests dynamiques des coupes, *etc*). Nous avons comparé ces résultats avec le temps passé dans le code de la politique de préchargement et finalement avec le temps global nécessaire à la récupération d'une page. Le tableau 6.2 présente ces mesures dans le cas où la page demandée par l'utilisateur est présente dans le cache (tableau 6.2(b)) et dans le cas où celle-ci doit être récupérée (tableau 6.2(a)). On constate que le temps passé dans le code de gestion des aspects n'excède jamais un millième du temps passé à l'exécution de la politique de préchargement.

Nous avons également mesuré le surcoût d'Arachne par rapport à l'implémentation statique de la même politique. Pour cela, nous avons modifié manuellement le code source de Squid pour lui ajouter la politique de préchargement décrite précédemment. Nous avons comparé les performances d'un Squid augmenté statiquement avec les per-

	Arachne Phase1	Manual Phase1	Arachne Phase2	Manual Phase2	diff Phase1 -Phase2
Bande passante (requêtes/s)	5.59	5.59	5.58	5.59	
Temps de réponse (ms)	1131,42	1146,07	1085,31	1074,55	1,2 % - -1%
Tps de réponse (page manquante) (ms)	2533,50	2539,52	2528,35	2525,34	0,2% - 1,8 %
Tps de réponse (page en cache) (ms)	28,96	28,76	30,62	31,84	-0,6 % - 3,8 %
Ratio de pages en cache	59,76	59,35	61,77	62,22	-0,6% - 0,7 %
Erreurs	0.51	0.50	0.34	0.34	-1,9 % - 0%

Tab. 6.3: Mesure de PolyMix-4 pendant les deux pics de requêtes

formances d'un Squid augmenté avec Arachne. Pour mesurer les performances de ces deux versions de Squid, nous avons utilisé l'outil d'évaluation Web Polygraph [RW04]. Après une mise en charge du cache, l'outil Polymix-4 permet de simuler 24 heures de trafic réseau, y compris les deux pics journaliers de requêtes observés sur les serveurs en production. Nous avons réalisé dix simulations de nos deux Squid. Leurs comportements pendant les pics de requêtes nous intéressent particulièrement : c'est pendant ces périodes que le cache est le plus chargé. Le tableau 6.3 présente les mesures obtenues pendant les pics. Le taux de requêtes et la bande passante sont mesurés depuis le côté client. Ces résultats nous montrent qu'il n'y a pas de différence visible de performance entre les deux versions du cache. En effet, les différences de temps de réponses entre les deux versions du cache n'excèdent pas les différences obtenues pour le même cache entre deux simulations.

6.3.2 Corrections de trous de sécurité

Dans le paragraphe 3.1, nous avons expliqué que l'apparition toujours plus rapide des virus informatiques nécessitait la mise en place de solutions aptes à répondre rapidement à cette menace. Nous nous proposons d'utiliser Arachne pour la correction à chaud de trous de sécurité. Pour cela, nous avons développé trois aspects. Chacun corrige un type de vulnérabilité bien connu des programmes C : les débordements de tableaux, les exploitations de l'allocateur mémoire et les exploitations de chaînes de format (les deux premiers ont été présentés pour illustrer le langage d'aspect d'Arachne dans le chapitre 4 respectivement dans les sections 4.4.4 et 4.4.1). Ces trois aspects agissent comme des patchs dynamiques. Ces patchs sont génériques : par exemple, le patch pour les débordements de tableaux peut s'appliquer à n'importe quel tableau.

Il n'est pas concevable pour des raisons de performance de corriger toutes les vulnérabilités potentielles d'une application. Un programme pouvant contenir un nombre important de tableaux, cela impliquerait un surcoût trop important à l'exécution. Ainsi, nos patchs doivent être configurés pour ne corriger que la vulnérabilité découverte et pour spécifier quelles actions entreprendre lorsqu'une attaque est détectée (ignorer, signaler, *etc*). Ainsi, lorsqu'un bulletin informant d'une vulnérabilité dans un logiciel libre est diffusé, un administrateur système peut en utilisant les informations contenues dans ce bulletins (logiciel vulnérable, type de vulnérabilité, localisation de la vulnérabilité) ou encore les sources du logiciel vulnérable, configurer un de nos patchs puis,

en utilisant Arachne, l'injecter à la volée.

6.3.2.1 Configuration des *patchs*

Comme nous le verrons plus loin, les informations nécessaires à la configuration de nos patchs sont dans la plupart des cas accessibles dans les bulletins annonçant les vulnérabilités. Nos patchs corrigeant chacun un type de vulnérabilité, les informations nécessaires à leur configuration diffèrent.

Dans le cas des débordement de tampons, il est nécessaire de renseigner l'aspect avec le nom du tableau C vulnérable. De manière similaire, dans le cas d'une exploitation de chaîne de format, le nom de la fonction à paramètres variables vulnérable est requis. Seule l'exploitation de l'allocateur de mémoire C ne requiert pas de configuration. En effet, comme nous l'avons vu dans la section 4.4.1 le correctif de cette vulnérabilité (listing 4.1) est complètement indépendant du programme de base.

En plus des informations requises pour la localisation des vulnérabilités, il est également possible de fournir des informations additionnelles pour affiner la localisation des vulnérabilités. Ainsi, l'administrateur peut fournir les noms de fonctions dans lesquelles sont réalisés : des accès hors des bornes d'un tableau, des appels inconsistants à une fonction variadiques ou à l'allocateur mémoire.

6.3.2.2 Faisabilité

Pour qu'une telle approche soit praticable, elle doit atteindre deux objectifs. Premièrement, elle doit être facile à mettre en œuvre pour les administrateurs système. Deuxièmement, l'injection dynamique des correctifs ne doit pas amputer les performances du programme corrigé.

Afin de vérifier la faisabilité de notre approche, nous avons analysé une partie des bulletins d'alertes concernant des logiciels C du site de la distribution Linux Debian [Deb]. Pour chaque bulletin correspondant à un débordement de tableaux, à une exploitation de l'allocateur mémoire ou à une exploitation de chaîne de format, nous avons mesuré le temps nécessaire pour découvrir les informations requises pour la configuration des *patchs* correspondant. Les sources d'information à partir desquelles nous avons travaillé sont les bulletins d'annonce des vulnérabilités et les sources des applications vulnérables. Nous avons volontairement omis les correctifs fournis puisque ceux-ci n'étaient pas nécessairement disponibles lors de la diffusion originale des bulletins de sécurité. De même, nous n'avons pas utilisé d'outils d'aide à la recherche de vulnérabilités. Nous avons considéré que même un administrateur consciencieux n'allouerait pas plus de vingt minutes à la recherche d'une vulnérabilité. La figure 6.2 résume le résultat de nos recherches.

Dans plus de 20% des alertes examinées, il n'y avait pas d'information permettant la localisation des vulnérabilités dans les bulletins. Il était alors impossible de configurer nos *patchs*. Néanmoins, bien que de tels bulletins ne soient d'aucune aide pour nous, c'est également le cas pour un éventuel pirate[2].

Parmi les bulletins restants, environ 16% font état du fichier dans lequel se situe la vulnérabilité. Cette information n'offre que peu d'aide : il est difficile de localiser

[2]Un bulletin annonçant qu'il existe une vulnérabilité dans un logiciel n'est pas vraiment une information !

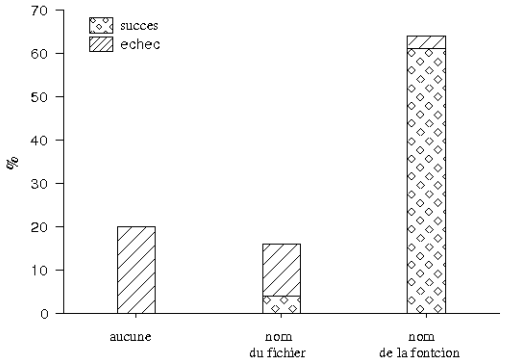


FIG. 6.2: Résultats des recherches d'information pour la localisation des vulnérabilités

une vulnérabilité manuellement dans des fichiers dont la taille peut parfois atteindre 8000 lignes comme cela est le cas pour le logiciel *sendmail*. Dans cette situation, nous n'avons pu identifier que les vulnérabilités de type chaîne de format puisque les fonctions variadiques demeurent assez rares même dans 8000 lignes de code.

Dans les 63% restants, l'information fournie par les bulletins est un nom de fonction dans laquelle se trouve une vulnérabilité. Dans ces situations, nous avons pu identifier avec succès toutes les vulnérabilités à l'exception d'une seule[3].

Dans tous les cas où nous avons pu localiser la vulnérabilité, nous l'avons fait en moyenne en huit minutes. Ces statistiques montrent clairement que l'approche que nous proposons ici est applicable. En effet, celle-ci permet de corriger plus de 65% des débordements de tableaux, exploitations de l'allocateur mémoire et exploitations de chaînes de format, en un temps convenable.

6.3.2.3 Mesures de performance

Nous avons montré que notre approche ne nécessitait qu'un investissement en temps de travail somme toute limité de la part d'un administrateur système. Bien que ce soit une condition incontournable, celle-ci n'est pas suffisante. Il est également indispensable que les performances du programme corrigé soient compatibles avec ses objectifs fonctionnels. Nous avons vu dans le paragraphe 6.2 que l'interception par un aspect d'un accès à un tableau pouvait entraîner un surcoût 10000 fois supérieur à son exécution native. Nous avons également mesuré que l'interception des appels à l'allocateur mémoire C est 5 à 6 fois supérieure à l'appel original et que la vérification d'un accès à un paramètre d'une fonction variadique est 2500 fois plus lente qu'un accès ordinaire. Ces coûts sont locaux et ils ne reflètent pas l'ensemble de l'exécution d'un programme. Aussi, nous pensons qu'ils sont largement ventilés sur l'exécution d'un programme.

Pour vérifier cette allégation, nous avons expérimenté nos *patchs* sur plusieurs des bulletins analysés précédemment. Nous présentons ici le cas du bulletin de sécurité *CVE-2004-0185* [UC04]. Ce bulletin de sécurité concerne le logiciel *wu-ftpd* (*Washington University File Transfert Protocol Daemon*). *wu-ftpd* est un serveur FTP largement

[3]En l'occurrence, nous avions bien identifié une vulnérabilité mais après vérification sur la version corrigée du logiciel il s'avérait que ce n'était pas celle identifiée par le bulletin de sécurité

	cycles	durée
ajout	16150	1,16μs
mise à jour	220	15,9ns

TAB. 6.4: Temps d'exécution du code de supervision de Squid (listing 4.6) pour une requête client

utilisé. Il a servi de base de développement à de nombreux autres serveurs FTP comme *BSD ftpd* et *ProFTPD* [Sip]. La large diffusion de *wu-ftpd* en fait un exemple réaliste pour nos évaluations.

La vulnérabilité identifiée par le bulletin *CVE-2004-0185* est un débordement de tableau dans la fonction *skey_challenge* du fichier *ftpd.c* et qui affecte les versions 2.6.0 à 2.6.2 du logiciel. Bien qu'il n'en existe aucune exploitation connue, cette vulnérabilité est critique et pourrait permettre la prise de contrôle à distance d'un serveur *wu-ftpd*.

Afin de comparer les performances de *wu-ftpd* dans sa version 2.6.2 avec cette même version corrigée dynamiquement, nous avons utilisé le logiciel *dkftpbench* [Keg], un outil permettant de charger un serveur FTP et de mesurer ses performances. *dkftpbench* simule par des automates, des clients se connectant et téléchargeant un fichier donné de 5Mo avant de se déconnecter. *dkftpbench* mesure le nombre de connections simultanées, le débit du serveur ou encore le délai de connection. Nous avons réalisé nos mesures entre deux machines connectée par un lien Ethernet à 100Mbit, la machine exécutant *wu-ftpd* est un Pentium 4 à 3,3GHz équipée de 1Go de mémoire RAM.

Nos mesures montrent que l'impact de l'injection d'aspects est négligeable. En effet, le nombre moyen de clients simultanément connecté est supérieur à un millier de clients dans la version originale et de cinq inférieur dans la version protégée. Le débit constaté sur le lien réseau demeure identique. On observe néanmoins un léger impact sur le délai d'identification au serveur. Lors de l'expérience les deux machines étaient reliées directement par un câble croisé, la distance parcourue sur un réseau réel aurait totalement amorti ce délai.

6.3.3 Supervision des accès aux systèmes de fichiers

Pour illuster la problématique de la supervision d'une application système, nous avons mesuré l'utilisation des disques durs par Squid pour chaque client en utilisant l'aspect présenté dans le chapitre 4 (section 6.3.1 listing 4.6). Ces aspects nécessitent le tissage de code dans Squid et dans le noyau Linux. Le tableau 6.4 montre la durée d'exécution du code de supervision pour chaque requête. Ce temps varie selon que l'on ajoute une association (lorsque l'on « découvre » une nouvelle adresse IP, c'est-à-dire la première requête d'un client) ou que l'on met à jour une association (c'est-à-dire les requêtes d'un client connu). En utilisant *Web Polygraph*, nous avons constaté une diminution inférieure à 5% de requêtes traitées pendant la supervision. Un impact somme toute limité considérant que cet aspect se situe à la fois en espace utilisateur et en espace noyau.

6.4 Conclusion

Dans ce chapitre, nous nous sommes proposés d'évaluer les performances de notre système de tissage à la volée. Pour cela, nous avons investigué sur plusieurs niveaux de granularité l'impact des aspects sur l'exécution d'une application. D'abord, nous avons mesuré avec précision les coûts de déclenchement des aspects pour chaque construction du langage d'aspect. Ces coûts s'avèrent très variables : de quelques cycles dans le cas des aspects sur les appels de fonctions à quelques milliers de cycles pour les accès à des variables. Ensuite, nous avons testé notre système sur des problématiques et des logiciels concrets. Ces expérimentations nous ont permis de constater qu'Arachne était de taille à attaquer les domaines variés de l'évolution logicielle que sont l'ajout de fonctionnalité, la correction de trous de sécurité et la supervision. Ces expérimentations nous ont également montré que les coûts des constructions du langage d'aspect étaient largement absorbés sur l'ensemble de l'exécution d'une application.

A conclusion is the place where you got tired of thinking.

Steven Wright

Chapitre 7

Conclusion

Bilan

Nous avons abordé au travers de cette thèse le problème de l'adaptabilité des systèmes informatiques patrimoniaux. Ces systèmes sont souvent soumis à des exigences de performances élevées, empêchant leur arrêt pendant leur mise à jour. De plus, la pérennité de leur code rend leur redéveloppement aberrant. Les nombreuses solutions proposées pour l'adaptabilité des systèmes informatiques ont ainsi atteint leurs limites. Le *refactoring* d'une application vers une implémentation plus adaptable requiert un effort important de conception dont les bénéfices sont souvent mitigés par la problématique de la décomposition dominante [OT00]. De plus, l'émergence toujours plus rapide des nouvelles technologies annihile toute anticipation des besoins futurs. Les performances recherchées par les développeurs d'applications système sont incompatibles avec l'utilisation d'environnements d'exécution dédiés qui offriraient une adaptabilité à posteriori. La mise à jour à chaud est réservée à des systèmes hautement performants et requiert souvent des solutions *ad-hoc* (réplication matérielle, balancement de charge). Aujourd'hui, seuls les systèmes de réécriture dynamique de code permettent l'adaptabilité sans anticipation. Leur complexité d'utilisation explique néanmoins la non généralisation de cette approche.

L'évolution par la programmation par aspects est une approche langage puissante qui facilite le travail des développeurs d'adaptations. Le tissage dynamique permet la mise en œuvre des adaptations non anticipées lors de l'implémentation du système de base. D'une part, cette approche retarde le moment de l'adaptation : il n'est pas nécessaire lors de la conception d'un logiciel de penser les interfaces qui seront potentiellement utiles à des adaptation futures. D'autre part, elle offre une facilité d'utilisation qui supplante toutes les approches existantes. L'implémentation des systèmes de tissage requiert souvent la modification de la chaîne de développement des applications et sont alors inadaptées aux systèmes patrimoniaux. Devant l'absence de systèmes de programmation par aspects pour le C proposant un tissage à la volée sans modification de la chaîne de conception et de déploiement, nous avons choisi d'implémenter un tel système.

Nous avons donc construit Arachne, un système de programmation par aspects utilisant la réécriture de code à la volée. Nous avons défini pour notre système, un langage d'aspect qui se base sur les concepts et la syntaxe du langage C. Principalement, notre

langage raisonne sur les appels de fonction et sur la manipulation des variables. Il permet également de raisonner sur l'exécution du programme : l'enchaînement séquentiel ou imbriqué d'appels de fonctions et/ou de manipulations de variables. Notre langage propose également des concepts concurrents : les coupes ne sont pas limitées à une application mais peuvent exprimer des interactions entre plusieurs applications et/ou le système d'exploitation.

Nous avons au travers de nombreux exemples, montré que ce langage offrait une expressivité suffisante pour attaquer des problèmes d'adaptation variés comme l'intégration d'une politique de préchargement dans un cache Web, ou comme la correction de trous de sécurité dans un serveur FTP.

Nous avons construit un tisseur d'aspect pour ce langage. Notre tisseur fonctionne sur plateforme Linux, Intel 32 bits. Il utilise la réécriture de code à la volée pour modifier des systèmes patrimoniaux. Les techniques d'injection et de réécriture utilisées ne nécessitent pas d'adaptation du processus de développement ou de déploiement des systèmes tissés. Notre tisseur permet l'injection d'aspects dans des applications utilisateur et dans le noyau Linux sans interruption de l'exécution. De plus, l'implémentation modulaire du système de tissage nous a permis de développer deux extensions ; l'une pour le tissage dans des programmes C++ ; l'autre pour intercepter les accès à des variables locales C.

Nous avons conduit de nombreuses évaluations de performance d'Arachne. D'abord, nous avons quantifié précisément pour chaque construction de notre langage, le coût de l'interception des événements de l'exécution d'un programme C. Nous avons ensuite confronté Arachne à des applications diverses et variées de l'évolution sur des systèmes réels. Par exemple, nous avons montré que la programmation par aspects était adaptée à la correction de trous de sécurité dans de nombreuses applications du cache Web Squid au serveur FTP wu-ftpd.

Nous avons donc évalué au travers de cette thèse qu'un système de programmation par aspects utilisant la réécriture de code à la volée pour le tissage, était adéquat à l'évolution des systèmes patrimoniaux. D'une part, nous avons montré que notre langage d'aspect autorisait l'expression concise d'adaptations en permettant aux développeurs de raisonner sur les concepts du programme à adapter. D'autre part, nous avons évalué avec précision les coûts induits sur l'exécution d'un programme du déclenchement des aspects. Ces mesures permettent aux développeurs d'adaptation d'anticiper l'impact des aspects sur les performances des systèmes. Enfin, nous avons montré dans le cadre d'adaptations concrètes sur des applications hautement performantes que l'utilisation de programmation par aspects avait un coût acceptable.

Perspectives

Nous avons présenté dans le chapitre 5, que les techniques de réécriture d'Arachne s'appuyaient sur les spécifications du langage C : ELF, POSIX et ABI. Ces standards décrivent les interfaces fournies par l'environnement d'exécution, les informations devant être maintenues lors de l'exécution d'un programme C et un certain nombre de conventions de code. Ce sont ces standards qui permettent à Arachne d'instrumenter du code indépendamment de la chaîne compilation (sous réserve que celle-ci respecte les standards susnommés). Il existe néanmoins d'autres informations intéressantes pour

un tisseur d'aspect. En effet, le fonctionnement des débogueurs nécessite que les exécutables embarquent des informations complémentaires permettant de réifier la relation entre le code machine et le code source. Ces informations sont spécifiées par les formats de débogage comme STABS ou DWARF. En termes de perspectives, l'analyse de ces informations ouvre des possibilités d'extension du langage d'aspect d'Arachne. Par exemple, il est envisageable de concevoir des coupes lors de l'accès en lecture/écriture d'un champs d'une variable structurée, ce qui est à priori impossible sans les informations de débogage. Nous avons franchi un premier pas dans cette direction puisque l'extension du système Arachne évoquée dans le chapitre 5, utilise les informations de débogage pour offrir le déclenchement d'aspects sur les accès à des variables locales.

Dans le chapitre 5, nous avions également évoqué une extension d'Arachne pour le tissage d'aspects dans des programmes C++. Cette extension n'en est qu'à un état préliminaire : aucun langage spécifique pour le C++ n'a été défini. Le processus de compilation d'un programme C++ maintient plus d'informations que le C, du fait de la résolution dynamique de types ou de l'héritage. Ces informations ouvrent des perspectives intéressantes pour la réalisation d'un langage d'aspect raisonnant sur les appels de méthodes, les accès à des membres de classes, ou encore sur les liens d'héritage.

Au cours de nos expérimentations, nous avons proposé des solutions pour des erreurs de programmation répandues : débordements de tableaux, exploitation de l'allocateur mémoire, chaîne de format. Ces erreurs sont connues comme vecteurs d'attaques informatiques. Nous avons également proposé une solution pour la détection d'interblocages de verrous. Il existe de nombreuses erreurs de conception, comme les « race conditions », les débordements d'entiers, *etc*, qui pourraient être résolues par une solution *aspect*. La conception d'une librairie d'aspects pour la détection d'erreurs de programmation offrirait une solution intermédiaire pour l'aide au développement d'applications.

Dans « Software Security Patches » [LSDM05b], nous avons proposé un outil pour la transformation de patchs traditionnels en aspects afin de les appliquer dynamiquement avec Arachne. Les « nouveaux » concepts de programmation comme la programmation par aspects, ayant souvent des difficultés à trouver un écho dans la communauté des développeurs, un tel outil présente un intérêt indéniable à la diffusion de la programmation par aspects. L'outil que nous avons créé à cet effet n'en est qu'au stade préliminaire et propose des défis intéressants. Notamment, il pose la problématique de l'applicabilité dynamique d'un patch « pensé statiquement ».

Nous avons introduit dans notre langage d'aspect, le placement des aspects sur plusieurs applications et la construction de séquences dont les aspects intermédiaires sont placés sur des applications différentes. Ces deux possibilités ne sont qu'un premier pas vers un langage d'aspect distribué. La conception d'un langage d'aspect distribué ouvre de nombreuses perspectives de travail autour d'Arachne. Du point de vue langage, on peut imaginer par exemple, des actions exécutées à distance de la coupe interceptée, ou encore des séquences synchronisées par des horloges logiques pour exprimer les interactions entre les agents d'un système.

Bibliographie

[AJ00] Martin Arlitt and Tai Jin. A workload characterization study of the 1998 world cup Web site. *IEEE Network*, 14(3) :30–37, May/June 2000.

[AL03] Iván Arce and Elias Levy. An analysis of the Slapper worm. *IEEE Security & Privacy*, 1(1) :82–87, January/February 2003.

[AS07] Bram Adams and Kris De Schutter. An aspect for idiom-based exception handling (using local continuation join points, join point properties, annotations and type parameters). In *Proceedings of the fifth Software-Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT'07)*, Vancouver, Canada, 2007.

[BCL+06] Éric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The Fractal component model and its support in Java. *Software – Practice & Experience*, 36(11–12) :1257–1284, September 2006.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo : A transparent dynamic optimization system. In *SIGPLAN'00 Conference on Programming Language Design and Implementation*, pages 1–12, Vancouver, BC, Canada, June 2000. ACM Press.

[Bel04] Fabrice Bellard. TCC – tiny C compiler's UNIX manual page, November 2004. `http://www.tinycc.org`.

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX 2005 Annual Technical Conference*, pages 41–46, Anaheim, CA, USA, April 2005. USENIX.

[BL76] Laszio Belady and Manny Lehman. A model of large program development. *IBM Systems Journal*, 15(3) :225–252, 1976.

[BRS+85] Robert Baron, Richard Rashid, Ellen Siegel, Avadis Tevanian, and Michael Young. MACH-1 : An operating system environment for large-scale multiprocessor applications. *IEEE Software*, 2(4) :65–67, July 1985.

[BSLR98] Noury M. Bouraqadi-Saâdani, Thomas Ledoux, and Fred Rivard. Safe metaclass programming. In *Proceedings of 1998 Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, pages 84–96, Vancouveur, BC, Canada, October 1998. ACM Press.

[BSP+95] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. *ACM Operating Systems Review*, 29(5) :267–284, 1995.

[CAI07] CAIDA. Code-red worms : A global threat. Technical report, The Cooperative Association for Internet Data Analysis, February 2007. `http://www.caida.org/research/security/code-red/`.

[Cas] Hugues Cassé. FrontC a frontend C in CAML including lexer, parser and pretty-printer.

[CDJ+89] Luca Cardelli, James E. Donahue, Mick J. Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 type system. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 202–212, Austin, TX, USA, January 1989. ACM Press.

[Cen03] Computer Emergency Response Team Coordination Center. Overview incident and vulnerability trends. Technical report, Carnegie Mellon University, 2003.

[CGRS90] Israel Cidon, Amit Gupta, Raphael Rom, and Christoph Schuba. Hybrid TCP-UDP transport for Web traffic. In *Proceedings of the 18th IEEE International Performance, Computing, and Communications Conference (IPCCC'99)*, pages 177–184, February 1990.

[CHL+98a] Charles Consel, Luke Hornof, Julia Lawall, Renaud Marlet, Gilles Muller, Jacques Noyé, Scott Thibault, and Eugen-Nicolae Volanschi. Partial evaluation for software engineering. *ACM Computing Surveys*, 30(3), September 1998.

[CHL+98b] Charles Consel, Luke Hornof, Julia Lawall, Renaud Marlet, Gilles Muller, Jacques Noyé, Scott Thibault, and Eugen-Nicolae Volanschi. Tempo : Specializing systems applications and beyond. *ACM Computing Surveys*, 30(3), September 1998.

[CKFS01] Yvonne Coady, Gregor Kiczales, Michael Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In Volker Gruhn, editor, *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01)*, volume 26, 5 of *Software Engineering Notes*, pages 88–98, Vienna, Austria, September 2001. ACM Press.

[CKO+02] Yvonne Coady, Gregor Kiczales, Joon Suan Ong, Andrew Warfield, and Michael Feeley. Brittle systems will break – not bend : Can aspect-oriented programming help ? In *Proceedings of the 10th workshop on ACM SIGOPS European workshop : beyond the PC*, pages 79–86, Saint-Emilion, France, September 2002. ACM Press.

[CM03] Huamin Chen and Prasant Mohapatra. CATP : A context-aware transportation protocol for HTTP. In *Proceedings of the 23rd In-*

ternational Conference on Distributed Computing Systems Workshops (ICDCSW'03), pages 922–927, 2003.

[Cor] Intel Corportation. Intel C/C++ compiler. `http://www.intel.com`.

[CY97] Ken-Ichi Chinen and Suguru Yamaguchi. An interactive prefetching proxy server for improvement of WWW latency. In *Proceedings of the INET'97 conference*, Kuala Lumpur, Malaysia, June 1997.

[Dar96] Erasmus Darwin. *Zoonomia, the Organic Laws of Life*. Saint Paul's church-yard, London, UK, 1796.

[Dar59] Charles Darwin. *On the Origin of Species by Means of Natural Selection*. John Murray, London, UK, 1859.

[Deb] Debian. Debian – security information. `http://www.debian.org/security/`.

[Des97] Solar Designer. JPEG COM marker processing vulnerability in Netscape browsers, 1997. `http://www.openwall.com/advisories/OW002-netscape-jpeg/`.

[DFL+05] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with Arachne. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, pages 27–38, Chicago, IL, USA, March 2005. ACM Press.

[DFL+06] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with Arachne. *Lecture Notes in Computer Science*, 1(1) :174–213, 2006. Transaction on Aspect-Oriented Software Development.

[DFS02] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *LLNCS*, pages 173–188, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[DG87] Jack W. Davidson and Joseph V. Gresh. CINT : A RISC interpreter for the C programming language. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 189–198, St. Paul, MN, USA, July 1987. ACM Press.

[ECM02] ECMA. *ECMA-334 : C# Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, second edition, December 2002. `http://www.ecma-international.org/publications/files/ecma-st/ECMA-334.pdf`.

[EF06] Michael Engel and Bernd Freisleben. TOSKANA : A toolkit for operating system kernel aspects. *Transactions on Aspect-Oriented Software Development II*, 4242 :182–226, 2006.

[EKO95] Dawson Engler, Frans Kaashoek, and James O'Toole Jr. Exokernel : an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 251–266, Copper Mountain, CO, USA, December 1995. ACM Press.

[ESV01] Andrew Edwards, Amitabh Srivastava, and Hoi Vo. Vulcan. Technical Report MSR-TR-99-76, Microsoft Research (MSR), January 2001.

[Foua] Free Software Foundation. GCC the GNU C compiler collection. `http://gcc.gnu.org/`.

[Foub] Free Software Foundation. GNU binutils. `http://www.gnu.org/software/binutils/`.

[FSLM02] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia Lawall, and Gilles Muller. Think : A software framework for component-based operating system kernels. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 73–86, Monterey, CA, USA, June 2002.

[GHSR07] Gilles Grimaud, Yann Hodique, and Isabelle Simplot-Ryl. On the use of metatypes for safe embedded operating system extension. *International Journal of Parallel, Emergent and Distributed Systems (IJPEDS)*, 22(1) :1–13, 2007. `http://www2.lifl.fr/POPS/Papers/rd2p.bib/grimaud-ijpeds-06`.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java^{TM} Language Specification*. Prentice Hall PTR, Indianapolis, IN, USA, third edition, June 2005.

[GK76] Adele Goldberg and Alan Kay. Smalltalk-72 instruction manual. Technical report, Learning Research Group, Xerox Palo, Alto Research Center, March 1976.

[GPRA98] Douglas Ghormley, David Petrou, Steven Rodrigues, and Thomas Anderson. SLIC : An extensibility system for commodity operating systems. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 39–52, New Orleans, LA, USA, June 1998. USENIX. `http://www.usenix.org/publications/library/proceedings/usenix98/ghormley.html`.

[GR85] Adele Goldberg and David Robson. *Smalltalk-80 : the language and its implementation*. Addison-Wesley, July 1985.

[Gro05] The TinyOS Working Group. TinyOS 2.0. In Jason Redi, Hari Balakrishnan, and Feng Zhao, editors, *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems, (SenSys)*, page 320, San Diego, CA, USA, November 2005. ACM Press.

[Gro06] Unix International Programming Languages Special Interest Group. DWARF debugging information format, revision 3.0.0. Technical report, Unix International, January 2006.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of micro kernel based systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP'97)*, pages 66–77, Copper Mountain, CO, USA, December 1997. ACM Press.

[IBCM00] Valérie Issarny, Michel Banâtre, Boris Charpiot, and Jean-Marc Menaud. Quality of service and electronic newspaper : The Etel solution. *Lecture Notes in Computer Science*, 1752 :472–496, 2000.

[Ins99] American National Standards Institute. *ANSI/ISO/IEC 9899-1999 : Programming Languages – C*. American National Standards Institute, New York, NY, USA, 1999.

[Int01] Intel Corportation. *IA-32 Intel Architecture Software Developer's Manual*. Intel Corportation, Mt. Prospect, IL, USA, 2001.

[JK97] Richard Jones and Paul Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In Mariam Kamkar, editor, *Proceedings of the Third International Workshop on Automatic Debugging*, volume 2, pages 13–26, May 1997.

[Jon96] Neil D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3) :480–503, September 1996.

[Keg] Dan Kegel. dkftpbench. `http://www.kegel.com/dkftpbench/`.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[KRB91] Gregor Kiczales, Jim Des Rivieres, and Daniel Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[Kue95] Geoffrey H. Kuenning. Kitrace : Precise interactive measurement of operating systems kernels. *Software – Practice & Experience*, 25(1) :1–21, January 1995.

[Lap90] Phillip A. Laplante. Heisenberg uncertainty. *ACM SIGSOFT Software Engineering Notes*, 15(5) :21–22, October 1990.

[LBH04] Oussama Layaida, Slim Benatallah, and Daniel Hagimont. A framework for dynamically configurable and reconfigurable network-based multimedia adaptations. *Journal of Internet Technology*, 5(4) :57–,–66, 2004. Special Issue on real time media delivery over the Internet.

[LBS04] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. Generic advice : On the combination of AOP with generative programming in AspectC++. In Gabor Karsai and Eelco Visser, editors, *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE '04)*, volume 3286 of *LNCS*, pages 55–74. Springer-Verlag, October 2004.

[LM07] Nicolas Loriant and Jean-Marc Menaud. Generalized dynamic probes for the Linux kernel and applications with Arachne. In *Proceedings of the 2007 IADIS Applied Computing International Conference (AC'O7)*, Salamanca, Spain, February 2007.

[LSDFM06] Nicolas Loriant, Marc Ségura-Devillechaise, Thomas Fritz, and Jean-Marc Menaud. A reflexive extension to Arachne's aspect language. In *Proceedings of 2006 AOSD workshop on Open and Dynamic Aspect Languages (ODAL'O6)*, Bonn, Germany, March 2006.

[LSDM05a] Nicolas Loriant, Marc Ségura-Devillechaise, and Jean-Marc Menaud. Server protection through dynamic patching. In *Proceedings of the 11th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'05)*, pages 343–349, Changsha, Hunan, China, December 2005. IEEE Computer Society.

[LSDM05b] Nicolas Loriant, Marc Ségura-Devillechaise, and Jean-Marc Menaud. Software security patches – audit, deployment and hot update. In *Proceedings of the 4th AOSD workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS'O5)*, pages 25–29, Chicago, IL, USA, March 2005.

[Mas02] Anthony Massa. *Embedded Software Development with eCos*. Prentice Hall PTR, Indianapolis, IN, USA, November 2002.

[MH99] Vlada Matena and Mark Hapner. Enterprise JavaBeans^{TM} specification, v1.1. Technical report, Sun Microsystems, Palo Alto, CA, 1999.

[MLD05] Gilles Muller, Julia L. Lawall, and Hervé Duchesne. A framework for simplifying the development of kernel schedulers : Design and performance evaluation. In *In Proceedings of the 2005 Conference on High Assurance Systems Engineering (HASE'05)*, pages 56–65, Heidelberg, Germany, October 2005. IEEE Computer Society.

[Moo00] Richard J. Moore. Dynamic probes and generalised kernel hooks interface for Linux. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, USA, October 2000. USENIX.

[Moo01] Richard J. Moore. A universal dynamic trace for linux and other operating systems. In *Proceedings of the FREENIX Track : 2001 USENIX Annual Technical Conference*, pages 297–308, Boston, MA, USA, June 2001.

[MR07] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *In Proceedings of the Third EuroSys Conference*, pages 327–340. ACM Press, March 2007.

[MTY05] Hidehiko Masuhara, Hideaki Tatsuzawa, and Akinori Yonezawa. Aspectual Caml : an aspect-oriented functional language. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 320–330, Tallinn, Estonia, September 2005. ACM Press.

[MvRT+90] Sape Mullender, Guido van Rossum, Andrew Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba : A distributed operating system for the 1990s. *IEEE Computer Society*, 23(5) :44–53, 1990.

[OT00] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In Mehmet Aksit, editor, *Proceedings of the Symposium on Software Architectures and Component Technology : The State of the Art in Software Development*, Twente, Netherlands, January 2000. Springer.

[PAB+95] Carlton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispen Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization : Streamlining a commercial operating system. In *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP'95), Operating Systems Review*, pages 314–324, Copper Mountain, CO, USA, December 1995. ACM SIGOPS.

[PAG03] Andrei Popovici, Gustavo Alonso, and Thomas R. Gross. Just-in-time aspects : Efficient dynamic weaving for Java. In *Proceedings of the 2nd international conference on Aspect-Oriented Software Development (AOSD'03)*, pages 100–109, Boston, MA, USA, March 2003. ACM Press.

[PCE+05] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating system problems using dynamic instrumentation. In *Proceedings of the 2005 Linux Symposium*, volume 2, pages 49–64, Ottawa, Canada, July 2005.

[PDFS01] Renaud Pawlak, Laurence Duchien, Gerard Florin, and Lionel Seinturier. Dynamic wrappers : Handling the composition issue with JAC. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS'01)*, pages 56–65, Santa Monica, CA, USA, September/October 2001. IEEE Computer Society.

[Pes00] David Pescovitz. Monsters in a box. *Wired*, 8(12), 2000.

[PKFH02] David J. Pearce, Paul H. J. Kelly, Tony Field, and Uli Harder. GILK : A dynamic instrumentation tool for the Linux kernel. *Lecture Notes in Computer Science*, 2324 :220–236, 2002.

[Pre04] Roger S. Pressman. *Software Engineering – A Practitioner's Approach*. McGraw-Hill, New York , NY , USA, sixth edition, 2004.

[PSD+04] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gérard Florin, Fabrice Legond-Aubry, and Laurent Martelli. JAC : An aspect-based distributed dynamic framework. *Software – Practise & Experience*, 34(12) :1119–1148, October 2004.

[PSDF01] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gérard Florin. JAC : A flexible solution for aspect-oriented programming in java. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION'01)*, pages 1–24, Kyoto, Japan, September 2001. Springer-Verlag.

[RAA+88] Marc Rozier, Vadim Abrossimov, François Armand, Michel Gien, Marc Guillemont, F. Hermann, Claude Kaiser, P. Leonard, S. Langlois, and W. Neuhauser. Overview of the Chorus distributed operating system. Technical Report CS/TR-88-7, Chorus Systèmes, Montigny-le-Bretonneux, France, June 1988.

[RL04] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*. Internet Society, February 2004.

[Ros03] Guido Van Rossum. *The Python Language Reference Manual*. Network Theory Ltd., Bristol, UK, September 2003.

[RW01] Michael Rabinovich and Hua Wang. DHTTP : An efficient and cache-friendly transfer protocol for Web traffic. In *INFOCOM*, pages 1597–1606, 2001.

[RW04] Alex Rousskov and Duane Wessels. High-performance benchmarking with Web Polygraph. *Software – Practice & Experience*, 34(2) :187–211, February 2004.

[SDML+06] Marc Ségura-Devillechaise, Jean-Marc Menaud, Nicolas Loriant, Thomas Fritz, Rémi Douence, Mario Südholt, and Egon Wuchner. Dynamic adaptation of the Squid web cache with Arachne. *IEEE Software*, 23(1) :34–41, 2006. Special Issue on Aspect-Oriented Computing.

[SDMML03] Marc Ségura-Devillechaise, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Web cache prefetching as an aspect : Towards a dynamic-weaving based solution. In *Proceedings of the 2nd international conference on Aspect-Oriented Software Development*, pages 110–119, Boston, MA, USA, March 2003. ACM Press.

[SESS96] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster : Surviving misbehaved kernel extensions. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, WA, USA, October 1996. USENIX Association.

[Sip] Jesse Sipprell. ProFTPD. `http://www.proftpd.org/`.

[SM04] Colleen Shannon and David Moore. The spread of the Witty worm. *IEEE Security & Privacy*, 2(4) :46–50, July/August 2004.

[Smi82] Brian Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, Massachusetts Institute of Technology, January 1982. `http://repository.readscheme.org/ftp/papers/bcsmith-thesis.pdf`.

[SMJ00] Matthew Smart, Robert Malan, and Farnam Jahanian. Defeating TCP/IP stack fingerprinting. In *Proceedings of the 9th USENIX Security Symposium*, pages 229–240, Denver, C0, USA, August 2000. USENIX.

[Sof07] TIOBE Software. Tiobe programming community index, September 2007. `http://www.tiobe.com/tpci.htm`.

[Spa94] Eugene Spafford. Computer viruses as artificial life. *Artificial Life*, 1(3) :249–265, 1994.

[SPLS+06] Wolfgang Schröder-Preikschat, Daniel Lohmann, Fabian Scheler, Wasif Gilani, and Olaf Spinczyk. Static and dynamic weaving in system software with aspectc++. In *Proceedings of the 39th Annual Hawaii International Conference on System Sciences, (HICSS'06*, volume 9, pages 214–223, Kauai, USA, January 2006. IEEE Computer Society.

[Sta95] Tool Interface Standard. Executable and linking format specification, revision 1.2. Technical report, TIS Committee, May 1995.

[SU94] UNIX System Laboratories System Unix. *System V Application Binary Interface Intel 386 Architecture Processor Supplement*. Prentice Hall Trade, 1994.

[Szy97] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley, December 1997.

[Tam01] Ariel Tamches. *Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels*. PhD thesis, University of Wisconsin Madison, USA, 2001.

[Teaa] The AspectSharp Team. AspectSharp : DotNet AOP framework. `http://sourceforge.net/projects/aspectsharp/`.

[teab] The ERESI team. The ERESI reverse engineering software interface. `http://www.eresi-project.org/`.

[teac] The FUSE team. FUSE : Filesystem in userspace. `http://fuse.sourceforge.net/`.

[TM99a] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 117–130, Berkeley, CA, USA, February 1999. USENIX.

[TM99b] Ariel Tamches and Barton P. Miller. Using dynamic kernel instrumentation for kernel and application tuning. *The International Journal of High Performance Computing Applications*, 13(3) :263–276, Fall 1999.

[Ubu05] Ubuntu. Squid proxy cache double memory free vulnerability. `http://www.security.nnov.ru/Idocument338.html`, April 2005.

[UC04] US-CERT/NIST. Buffer overflow in the skey_challenge authentification for wu-ftp daemon 2.6.2, May 2004. `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2004-0185`.

[VSCF05] Wim Vanderperren, Davy Suvée, María Agustina Cibrán, and Bruno De Fraine. Stateful aspects in JAsCo. In Thomas Gschwind, Uwe Aßmann, and Oscar Nierstrasz, editors, *Proceedings of 2005 Conference Software Composition*, volume 3628 of *Lecture Notes in Computer Science*, pages 167–181, Edinburgh, Scotland, April 2005. Springer.

[WK03] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, pages 149–162, San Diego, CA, USA, February 2003.

[YKC06] Yoshisato Yanagisawa, Kenichi Kourai, and Shigeru Chiba. A dynamic aspect-oriented system for OS kernels. In Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, editors, *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06)*, pages 69–78, Portland, OR, USA, 2006. ACM Press.

[Yok92] Yasuhiko Yokote. The Apertos reflective operating system : The concept and its implementation. In Andreas Paepcke, editor, *OOPSLA'92 Conference Proceedings : Object-Oriented Programming Systems, Languages, and Applications*, pages 414–434. ACM Press, 1992. `http://www.acm.org/pubs/articles/proceedings/oops/141936/p414-yokote/p414-yokote.pdf`.

Glossaire

COFF

COFF, « Common Object File Format », est un format de fichier pour les exécutables ensuite étendu aux librairies partagées. Introduit sous Unix , il a ensuite été remplacé par le format ELF. COFF présente de nombreuses limitations, notamment sur les informations de débogage qui motivèrent deux variantes du format : XCOFF et ECOFF. COFF est à l'origine du format PE, « Portable Executable » utilisé sous Windows NT.

DWARF

DWARF, « Debugging With Attributed Record Formats » est un format de fichier décrivant les informations de débogage pour des langages procéduraux (principalement C, C++ et FORTRAN) permettant le débogage au niveau source. DWARF est extensible et indépendant de l'architecture. Il est principalement utilisé conjointement à l'ELF dans des environnements UNIX. La spécification en est aujourd'hui à sa troisième version [Gro06].

ELF

ELF, « Executable and Linking Format », (à l'origine l'acronyme signifiait « Extensible Linking Format », est un format de fichier pour les exécutables et les librairies partagées [Sta95]. Conçu en remplacement des formats *a.out* et COFF, ELF est le standard utilisé sur un nombre important de systèmes : Linux, Solaris, BSD, ou encore OpenVMS.

Zero-day worm/virus

Un *zero-day worm/virus* est une attaque informatique exploitant une faille de sécurité le plus rapidement possible après sa découverte. La stratégie appliquée par l'attaquant est de prendre de vitesse les acteurs de la défense informatique en exploitant la faille découverte avant la conception et la mise en place d'un protocole de contre mesures. *Sasser* et *Witty* sont deux exemples de vers de type *zero-day*.

Buffer Overflow

Un débordement de tableau est une condition anormale du fonctionnement d'un programme dûe à un accès mémoire invalide et provoquant le plus souvent la terminaison du programme. Cette état anormal du système peut permettre à un

utilisateur malveillant de prendre le contrôle du système hôte. L'accès mémoire invalide est provoqué par l'écriture au delà des bornes d'un tableau de taille fixe. Les données supplémentaires viennent alors écraser les données adjacentes au tableau. Dans le cas des programmes C, les tableaux stockés sur la pile côtoient les adresses de retour des fonctions et qui une fois modifiées peuvent permettre de déclencher l'exécution de code arbitraire.

Double Free Bug

Un *Double free bug* est une condition anormale du fonctionnement de la libraire standard C provoquée par la désallocation d'un bloc mémoire non alloué. Cette erreur de programmation peut être exploitée à des fins malveillantes pour déclencher l'exécution de code arbitraire.

Pipeline d'instructions

Un pipeline est une technique de conception des processeurs où l'exécution des instructions est découpée en étages, et où à un instant donné, chaque étage peut exécuter une instruction.

Social engineering

L'ingénierie sociale est un terme regroupant les activités, techniques et tactiques visant à obtenir une chose (comme par exemple un mot de passe ou une autorisation d'accès) en exploitant la confiance, l'ignorance ou la crédulité de personnes. Une de ces techniques, le *Phishing* ou hameçonnage en français consiste à usurper l'identité d'un tiers de confiance comme par exemple une banque, afin d'obtenir une information par exemple le code d'accès d'un client.

Cache Web

Un cache est un matériel ou un logiciel intermédiaire qui duplique des données dont les originaux présentent un coût important de récupération ou de calcul. C'est donc un stockage intermédiaire permettant l'accès rapide à des données fréquemment accédées. Les caches Web répliquent le contenu Web récemment accédé afin de diminuer la latence perçue par les utilisateurs d'un réseau informatique.

Reverse engineering

La rétro-ingénierie est un terme regroupant les procédés permettant de retrouver les origines d'un objet ou d'un système grâce à une analyse abductive de ces fonctionnalités et de sa structure. En informatique, la rétro-ingénierie d'un logiciel à pour objectif de créer une représentation abstraite de ce logiciel ou de reconstituer les sources de celui-ci, on parle alors parfois de *décompilation*.

OS fingerprinting

La prise d'empreinte de systèmes d'exploitation regroupe les techniques permettant de déterminer le système d'exploitation utilisé par une machine distante. La plus technique la plus connue consiste à observer les délais et le nombre de

retransmissions des paquets TCP perdus lors d'une communication. En effet, les valeurs par défaut de ces paramètres varient selon les systèmes d'exploitation.

Évolution dynamique des systèmes d'exploitation, une approche par la programmation par aspects

Nicolas Loriant

Dans un contexte où les technologies de communication évoluent à grande vitesse, la course effrénée à l'intégration de nouvelles fonctionnalités dans les systèmes informatiques est souvent engagée au détriment de solutions stables, extensibles et adaptables, et par conséquent plus pérennes.

La réalisation de systèmes adaptables a fait l'objet de nombreuses recherches ayant abouties à des solutions originales. Néanmoins, les propositions reposant sur des architectures adaptables et extensibles se révèlent souvent complexes, peu performantes et rigides ! En effet, il est difficile voir impossible d'anticiper quelles seront les évolutions futures, les interfaces permettant l'extensibilité se révèlent souvent inadaptées aux besoins réels. Contrairement aux architectures extensibles, les solutions permettant la transformation à la volée d'un système ne sont pas limitées par des choix architecturaux et s'avèrent plus performantes. Néanmoins, la complexité de ces approches les limitent à des utilisateurs experts.

Dans cette thèse, nous nous sommes fixés comme objectif de réconcilier adaptabilité dynamique, performances des systèmes informatiques et simplicité de programmation des évolutions. Pour cela, nous avons combiné deux approches : la réécriture dynamique de code pour fournir les mécanismes performants de transformation de programme ; et la programmation par aspects comme moyen d'expression des évolutions logicielles. Nous démontrons la validité de notre approche par des évaluations exhaustives de l'utilisation de notre prototype en réalisant des évolutions concrètes et variées de systèmes informatiques patrimoniaux.

Mots-clés : Évolution des systèmes informatiques, Programmation par aspects, Réécriture de code à la volée

With the never ending evolution of communication technologies and of multimedia, the race to integrate new functionalities has lead to a situation where systems are poorly adaptable. The design of extensible software is often difficult and often impact on software performances. Thus, many developers choose to simply distribute software patches that requires users to restart their systems. This situation is not only a disturbance for user but contribute to virus propagation.

In this thesis, we try to conciliate software extensibility and performances. Our approach consists in using an aspect oriented system based on binary code rewriting. Binary code instrumentation allows fine-grained code modification without impacting on performance, while aspect oriented programming helps developers to easily express adaptation code. We evaluate the evolution of systems using our aspect oriented framework : Arachne. We evaluate our approach following two axis : the use of aspect language to concisely express adaptation code, and performance impact of binary rewriting code.

Keywords : Software evolution, Aspect oriented programming, On-the-fly binary code instrumentation